AD-A090 623 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH F/G 9/2

DEVELOPMENT OF A HIGH LEVEL LANGUAGE AND CROSS-COMPILER FOR THE--ETC(U)

JUN 79 W W HATCHER

UNCLASSIFIED AFIT-CI-79-161T NL

1 OF 1

AD A090623

79-161T

Master's thesis

DEVELOPMENT OF A HIGH LEVEL LANGUAGE AND CROSS-COMPILER FOR THE INTEL 8080 MICROPROCESSOR

William Ward Hatcher

7 Jun 79

96

AFIT-CI-79-161T

Certificate of Approval:

J. S. Boland, III Associate Professor Electrical Engineering	J. R. Heath, Chairman Associate Professor Electrical Engineering
B. D. Carroll Associate Professor Electrical Engineering	Paul F. Parks, Dean Graduate School

012200

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 79-161T ✓	2. GOVT ACCESSION NO. AD-A090 623	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Development of a High Level Language and Cross-Compiler for the Intel 8080 Microprocessor		5. TYPE OF REPORT & PERIOD COVERED Thesis 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) William Ward Hatcher		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT Student at: Auburn University, Alabama		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433		12. REPORT DATE 7 June 1979 13. NUMBER OF PAGES 95
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASS 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) APPROVED FOR PUBLIC RELEASE AFR 190-17. Fredric C. Lynch FREDRIC C. LYNCH, Major, USAF Director of Public Affairs		23 SEP 1980
18. SUPPLEMENTARY NOTES Approved for public release: IAW AFR 190-17 Air Force Institute of Technology (ATC) Wright-Patterson AFB, OH 45433		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Attached		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DEVELOPMENT OF

A HIGH LEVEL LANGUAGE AND CROSS-COMPILER

FOR THE INTEL 8080 MICROPROCESSOR

William Ward Hatcher

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Science

Auburn, Alabama

June 7, 1979

DEVELOPMENT OF

A HIGH LEVEL LANGUAGE AND CROSS-COMPILER

FOR THE INTEL 8080 MICROPROCESSOR

William Ward Hatcher

Signature of Author

Date

Copy sent to:

Name	Date

VITA

William Ward Hatcher, son of Gatewood Matthews and Amy (Vaughan) Hatcher, was born May 18, 1944, in York, Alabama. He attended Sumter County Public Schools and graduated from Sumter County High School, York, in 1962. In September, 1962, he entered Livingston University and received the degree of Bachelor of Science (mathematics) in December, 1966. In January, 1974, he entered graduate school at Auburn University in Montgomery and received the degree of Master of Public Administration in August, 1976. In September, 1976, he entered graduate school at Auburn University and began work toward his Master of Science degree in Electrical Engineering. He married Diane, daughter of Sherman Alfred and Mary (Drake) Harris in November, 1966. They have one daughter, Stephanie Elizabeth and one son, William Todd.

THESIS ABSTRACT

DEVELOPMENT OF A HIGH LEVEL LANGUAGE AND

CROSS-COMPILER FOR THE INTEL 8080 MICROPROCESSOR

William W. Hatcher

Master of Science, June 7, 1979
(B.S., Livingston University, 1966)
(M.P.A., Auburn University in Montgomery, 1976)

95 Typed Pages

Directed by J. R. Heath

This paper describes the development of a high level language and cross-compiler written in BASIC PLUS for the Intel 8080 microprocessor. The language is a general purpose language which can be entered online on a DEC PDP-11/40 minicomputer and cross compiled for the I8080. The cross-compiler follows the general pattern of most high level language compilers and consists of a scanner, a parser, semantic routines and code generation procedures. The cross-compiler accepts the input language and produces assembly language which is provided to a cross-assembler to generate machine code either on paper tape or a disk file. This code is then loaded into the I8080 for execution.

The basic advantage of the high level language and cross-compiler is that it provides a user the capability to develop a program without having to know the details of microprocessor assembly language. Also,

program development aids available on the DEC PDP-11/40 minicomputer with an RSTS/E operating system are made available to the microprocessor programmer.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

Since the advent of the microcomputer in 1971, a new dimension in computing capability has been opened in the world of electronics. The microcomputer is a relatively inexpensive computer which, when compared to machines of 15 to 20 years ago, provides a tremendous amount of computing power for the dollar and has revolutionized the world of automation. However, one of the limitations of the microcomputer has been the limited availability of high level languages. Consequently, most programming is accomplished through assembly language. The use of assembly language is popular because of the obvious relation between the source and the object code. Also, memory and instruction usage is entirely under the control of the programmer and macros enable standard code sequences to be written and debugged once. However, with the rapidly falling price of memory and processors and the high cost of assembly language level software development, there is increasing pressure to use more economical methods of code generation since when in assembly language, much time must be spent in using the detail procedures of the language; especially in the relatively weak microcomputer assembly languages.[1] Additionally, interpretative languages such as BASIC do not offer the best solution since they execute slower than compiler generated code. Therefore, high level languages offer an attractive alternative to machine or assembly languages.

One of the primary advantages of a high level language is the lack of the requirement that the programmer needs to know the architecture of the machine. It is the compiler which handles registers, storage allocation and data conversions.[2] Also, symbolic variables increase the readability of the program; programmer productivity is increased; documentation is improved through a more understandable program; maintenance, modification and debugging are facilitated; and transportability is improved. As with most other things, there are always drawbacks. Primary among these is the additional memory required with high level languages (although with lowering costs it is not a important as it once was). Also, if a language is not properly suited for the purpose, the language may become a liability rather than an asset.[3] However, with all factors considered, there is a definite need for the development of high level languages for microcomputers.

This thesis will discuss the development of such a language (AU78) and cross-compiler for programming an Intel 8080 based microcomputer. This high level language is designed to provide the capability for accomplishing basic computing operations to the programmer through the use of the PDP-11/40 RSTS/E on-line time-sharing operating systems. It was designed with a combination of BASIC and COBOL type statements and to be on a par computationally with languages such as Tiny BASIC.[4] The major purpose of the language is to provide an easier method of programming the I8080. It offers the unique capability of developing a program directly on-line through the PDP-11. The cross-compiler will accept AU78 source code, which will be compiled into I8080 assembly language,

which in turn is passed to a cross-assembler to generate machine language on a paper tape or a disk file. Figure 1 depicts the operational flow process.

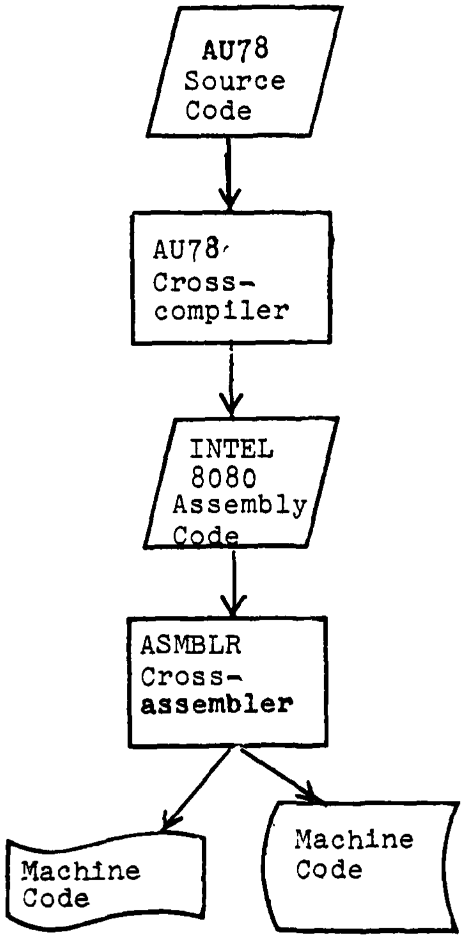

Figure 1. Operational flow process.

This is a very significant feature in that a programmer can very quickly write a short program to run on the I8080. Further, AU78 is designed in a modular manner such that it can be expanded to include additional specific requirements by adding subroutines for that purpose.

Another reason for the design and development of the language was so the author could gain a greater understanding and knowledge of the problems and trade-offs encountered when developing high level languages for microcomputers and subsequent cross-compilers for execution on minicomputers with time-share operating systems.

General Features of AU78

AU78 provides a variety of features to the programmer which are common to most higher level languages available on larger computer systems. The first to consider would be the input/output features. The AU78 compiler accepts input values from $+32,767_{10}$ to $-32,768_{10}$. Multiple values can be input through the calling of the input subroutine and values can either be used immediately or stored for later use. The output consists of either variable values or strings with up to 130 characters for use with a full line printer. The language provides for assigning specific values to variables or if it is not specified, the value of Ø will automatically be assigned. The computational operations include multiplication, division, addition, and subtraction and the use of parenthesis to allow for multiple levels of operations. In all equations, the proper order of precedence is insured. Conditional statements allow expressions or variables or integers or a combination of them all to be evaluated to determine if one value is greater than, less than, greater

than or equal to, less than or equal to, not equal to or equal to another. The results of these comparisons will be incorporated into the logical operation of an IF-THEN-ELSE type statement. In the IF-THEN-ELSE statement, the ELSE is optional and the next sentence will be executed if it is not present and the THEN condition fails.

When coding a program, line numbers are optional and statements may be labeled to provide for branching. The language further provides for transfer of data internally and transfer of control through branching or calling subroutines. Data to be used in subroutines is made available through variable storage areas.

General type statements include the following:

READ variable

WRITE variable

WRITE string

IF (expression-variable-integer) condition (expression-variable-integer) THEN statement ELSE statement

MOVE (variable-integer) TO variable

GOTO label

GOSUB label

RETURN

COMPUTE expression = variable

The major limitations of AU78 include the ability to rapidly input large volumes of data; although with additional subroutines, both magnetic tape, cards and disk could be used for input and output. The AU78 cross-compiler cannot handle floating-point numbers or double-precision arithmetic operations. Floating-point was investigated as a viable

capability for inclusion in AU78 but was not included for several reasons. The first was because the I8080 has no built-in hardware provisions for floating-point which would be required for efficient implementation. To offer any advantages over the implemented fixed-point number format, a thirty-two bit number (one sign bit, seven exponent bits and twenty-four mantissa bits) would be required for the representation of each operand. In the I8080, this would require four eight bit registers per operand leaving only two registers plus the accumulator. Since most all computations must be accomplished in the accumulator and only one full operand at a time can fit in the working registers of the I8080, the number of data manipulations and memory transfers required by simple double operand arithmetic functions make the execution of floating-point arithmetic function very slow and inefficient except in the case of special purpose applications. For example, to execute a simple two operand addition would require approximately one hundred program steps. Multiplication and division would likewise require an excessive number of program steps and a proportional increase in execution time. Lengthy, complex mathematical operations could not be accomplished with any real speed.[6] Finally, a smaller sixteen bit floating-point number format could have been developed more effectively, but the precision of the resulting operand values would have been no greater than what was available using an integer format.

Another limitation exists in string manipulation in that concatenation of strings cannot be accomplished. It should be noted that in

the existing advanced languages for microcomputers such as PL/M, PL/M68000, Tiny BASIC and FORT/80[7] many and even more of these same limitations exist.

Practical Applications

As stated previously, one of the main reasons for developing AU78 was to provide an easier method of programming the I8080 and to provide this capability through the PDP-11/40 RSTS/E operating system, complete with its full editing capabilities. One of the most common applications envisioned for the languages was to provide users a handy, readily available mechanism for writing general purpose programs for the I8080 without having to fully understand the assembly language of the I8080. Rather than spending time learning the details of the assembly language, the user could spend that time in problem solving.[8] This also holds true of even the experienced I8080 programmer and can save much time and effort especially considering the poor diagnostics associated with the available assemblers and cross-assemblers. A user can write a program and create a paper tape or a disk file and load the program directly into the I8080 and operate it in an on-line environment in a problem solving manner. The language as currently designed is especially good at applications where computations are performed based on input data. Examples of this might include interest rate calculations, income tax computations, budgetary projections and various other general business type applications.

Another major consideration in the development of the language and compiler is for use by the Air Force. The Air Force has recently begun

acquiring PDP-11s and is moving into the use of microcomputers. At the author's assignment at the Air Force Data Systems Design Center, a new PDP-11 has been acquired. The Air Force-wide policy is to program all machines as much as possible in a high level language. Because of the turnover of personnel, the high level language is easier to learn and document. The author's approach will be to interest the Air Force in AU78 or a modified version to allow early development of microcomputer systems without having programmers undergo extensive training in I8080 assembly language programming. The basics of AU78 could be learned by experienced programmers in a very short period of time. Also, with the modular approach used in the AU78 compiler, specific capabilities required by the Air Force could be easily added. The Air Force policy on standard systems is to develop systems for all computers at one agency in a high level language and send programs out to other users in machine language to prevent unauthorized changes. This language would provide that opportunity since there are few other microcomputer languages available. It is understood that the language would probably be changed or capabilities added for specific type requirements, but AU78 would provide an adequate base language for Air Force use. The use of microcomputers in the Air Force is already large, but they are used primarily in scientific and process control type operations. The future use of micros in a more business type orientation is unlimited. AU78 is a step in that direction and provides a basis for future growth and development.

Development of AU78 Routines

The remainder of this thesis will discuss the modules required in the development of the cross-compiler. As an overview to a compiler, there are several basic modules required and although they may be referred to in different terms or broken into fewer or more modules, the basics of a compiler include a scanner, a parser, semantic routines, and a code generator. However, the first step in the writing of a compiler is to define the language in terms of a grammar which defines valid sentences, operations, operators, and symbols. Table 1 describes the AU78 grammar. These entries are used to create a table which is used by the compiler to determine proper sentences. The scanner is a routine which scans the input character string and builds the output symbols of the programs, including integers, identifiers, reserved words, delimiters and operators. These are then passed to the parser which checks the symbols against the language definitions to determine if the sentences are properly constructed. If not, an error will be indicated. With the receipt of a valid input, a final pass is made to incorporate the symbol table into the code and the output code is written. Each of these routines will be discussed in detail to show how the cross-compiler was written. The entire compiler was written in a modular manner such that each routine for each type sentence is generally a separate set of code in the program. This allows for easy manipulation and addition of features in the language if future requirements dictate a particular type application not presently available. A complete user's manual and cross-compiler program source listing are provided in appendices A and B respectively.

Table 1. AU78 grammar.

```
<GRAMMAR> ::= <LINE> ( <GRAMMAR> /E)/ STOP

<LINE> ::= <STATEMENT> (; <LINE> /E)

<STATEMENT> ::= <MOD STATE> / <IMP STATE> /
                <LABEL STATE>

<MOD STATE> ::= IF <CONDITION> THEN <IMP STATE> (ELSE <IMP  STATE> )

<CONDITION> ::= <EXP><OP><EXP>

<OP> ::= </<>/>/<=/>=/=>

<IMP STATE> ::= READ <PARM> /WRITE <PARM> /
                WRITE <STRING> /GOTO <VAR> /
                GOSUB <VAR> /MOVE <VAR> TO <VAR> /
                RETURN/COMPUTE <EXP> = <VAR>

<PARM> ::= <VAR>

<LABEL STATE> ::= <VAR> ::<STATE> / <STRING>

<VAR> ::=<LETTER> (<LETTER> / <INT> /E)

<EXP> ::= <TERM> ( <AOP><EXP> /E)

<TERM>::= <FACTOR> ( <MOP><EXP> /E)

<FACTOR> ::=( <EXP> )/ <INT> / <VAR> / <STRING>

<AOP> ::=+ / -

<MOP> ::= * / /

<LETTER> ::=A/B/.....12

<INT> ::= <DIGIT> ( <INT> /E)

<DIGIT> ::= Ø/1/....19

<STRING> ::= "Any Character"
```

II. THE SCANNER

Perhaps the most important part of the cross-compiler is the scanner routine. It is in this phase that the input program is scanned sequentially and the basic elements or tokens of the program are identified. These include terminal symbols such as literals, variables, operators, and key words. Typically the source string of the program is converted into another string of symbols containing attributes of each basic element. These symbols are generally of a fixed size and consist of the elements syntactic class and a pointer to the table entry of the associated basic element. These symbols are used in later processing by other phases of the cross-compiler. Because the symbols are of fixed size, converting to them makes the later phases of the compiler operation easier to design.[12]

Included in this symbolic internal representation is a number which represents an identifier, integer, delimiter, key word, or operator. That is, all identifiers have the same internal number to represent them, as do each of the other terminal symbols. However, the terminal itself is needed by the parser, so it too must be stored for later use by the parser. The solution is to output two values. The first is the internal representation and an index to its position in the table. The second is the actual value itself.

In AU78, the above approach to the design of a cross-compiler was followed. Each key word, operator and delimiter was loaded into a table with an index to the appropriate entry. Each was given a symbolic value for the entire class. For example, all delimiters are classified as a 4, and key words as a 3. The scanner will scan an input sentence and break apart the sentence into its relative parts. Each part will then be assigned a value, and an index within the table and this information will be passed on to the parser. Figure 2 is a flowchart depicting the scanner processes and the interface links to the parser. Table 2 contains a list of key words, delimiters and operators.

A problem that had to be overcome was to be able to differentiate between alphabetic variables and key words. To accomplish this each character is read and stored until either a space, delimiter or numeric value is reached. Variables can contain integers, although they must begin with letters. If an integer is encountered in the scan, then the input is treated as a variable since key words are all alphabetic. If no integers are found, the input value is compared with the entries in the key word table. If a match is found, the indexed position in the table is saved and the value for a key word is saved along with the actual value of the key word and passed on to the parser. The same process is true of variables except that an index for the variable is not required since it is a unique symbol.

In the case of delimiters and operators, a similar process is followed where a table search takes place to insure the value is a valid, acceptable delimiter or operator for the language. If valid,

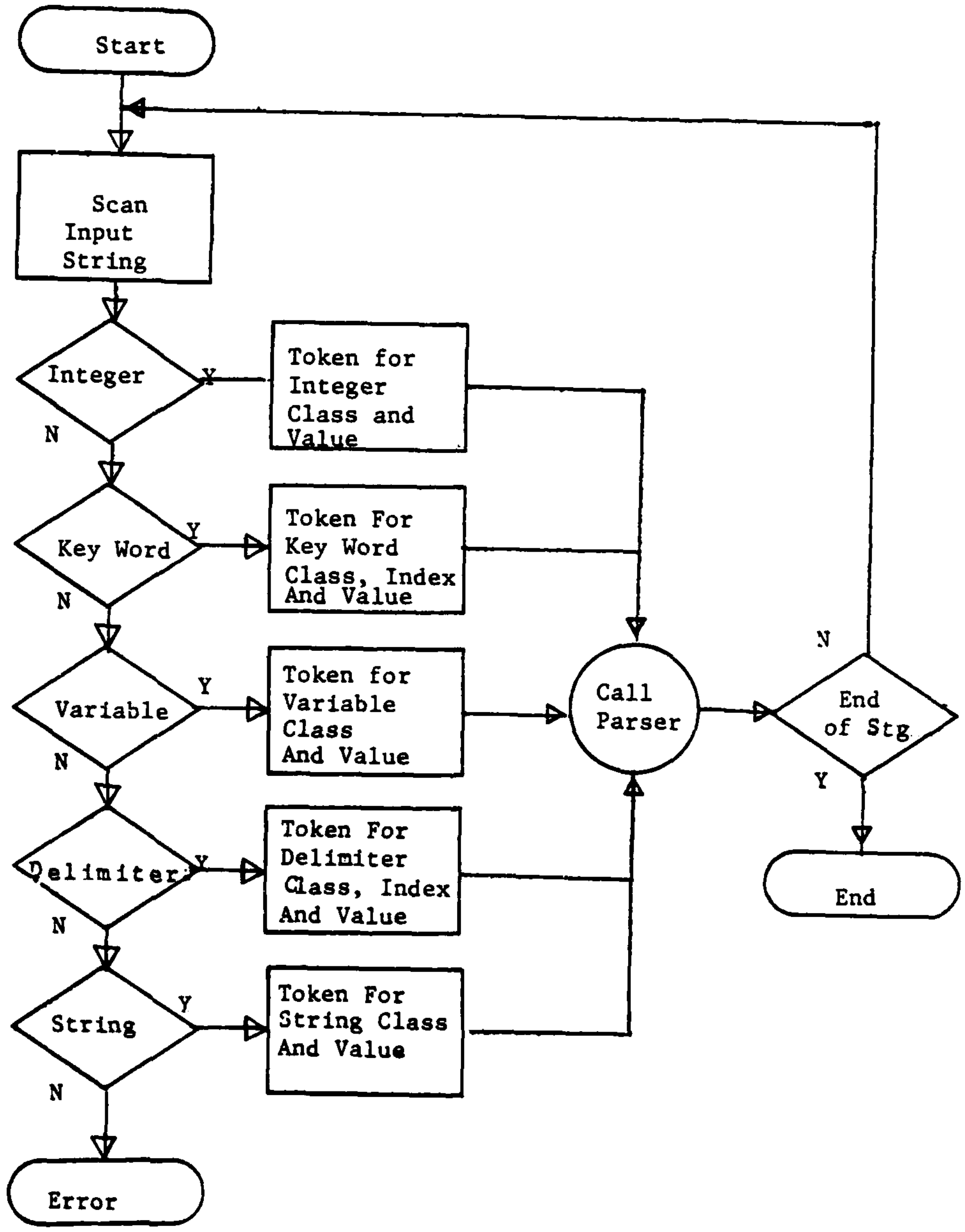

Figure 2. Flowchart of scanner.

Table 2. Key word, delimiter and operator list.

Key Word	Delimiters/Operators
IF	>
THEN	<
READ	=
WRITE	:
GOTO	;
GOSUB	+
MOVE	-
RETURN	*
ELSE	/
STOP	'
TO	<=
COMPUTE	>=
END	<>
	(
	)

Table 3. Example scan of a sentence.

STRT: IF 1Ø1 <SUM THEN MOVE 1Ø1 to STORE

Value	Class	Index in Class
STRT	2 (Variable)	0
:	4 (Delimiter)	4
IF	3 (Key Word)	1
101	1 (Integer)	0
<	4 (Delimiter)	2
SUM	2 (Variable)	0
THEN	3 (Key Word)	2
MOVE	3 (Key Word)	7
101	1 (Integer)	0
TO	3 (Key Word)	11
STORE	2 (Variable)	0

again a value for the class is stored and along with the index, the actual value is passed to the parser. Numeric values are checked to insure they contain only integers. A class value is then assigned for the integers, but an index is not required since a table of integers is not needed. Finally, the scanner will recognize a string value. The scanner will accept anything that is set apart by single quotes and assign a class value for a string.

Table 3 portrays an example sentence and the symbolic values that would be assigned and passed to the parser. As previously stated, these class values represent the type element and the index within the table of key words, delimiters and operators. As shown, variables and integers do not require indexes since they are unique.

One additional major function of the scanner is to detect initial errors in the input program. The edits at this point are primarily concerned with the various elements of a sentence and generally check to insure that input elements are acceptable and in correct form. Any element that does not fit into any of the various classes will be rejected as an error. It is impossible to detect at this point if an error was made in procedure. For instance, if a key word and an integer were run together without any separation, the two would be treated as a variable. As an example, if MOVE 1Ø to ADDR were written MOVE1Ø to ADDR, the MOVE1Ø would be treated as a variable and not as a key word and integer. It remains for the parser to detect errors of this nature.

III. THE PARSER

Once the input program has been broken down into symbols or tokens, the cross-compiler must recognize the phrases (syntactic construction) and interpret the meaning of the constructions. Each phrase is a semantic entity and is a string of tokens that has an associated meaning.

The first of these two steps is concerned solely with recognizing and thus separating the basic syntactical constructs in the source program. It also notes syntactic errors. Once the syntax of a statement has been ascertained, the second step is to interpret the meaning (semantics).[14] This is accomplished through the use of rules or reductions to build a derivation of the sentence. This is often performed through what is called a syntax tree. Figure 3 portrays the syntax tree derivation of a sample sentence in the AU78 language. The approach used here and in AU78 is called a top-down approach. A top-down parser builds the tree starting from the root and works downward to the terminal elements or nodes. In the example of Figure 3, GRAMMER is the root and a terminal node would be STRT. If the entire sentence can be parsed from the root and all nodes reached, then it is a valid sentence.

STRT: IF 101 < SUM THEN MOVE 101 TO STORE

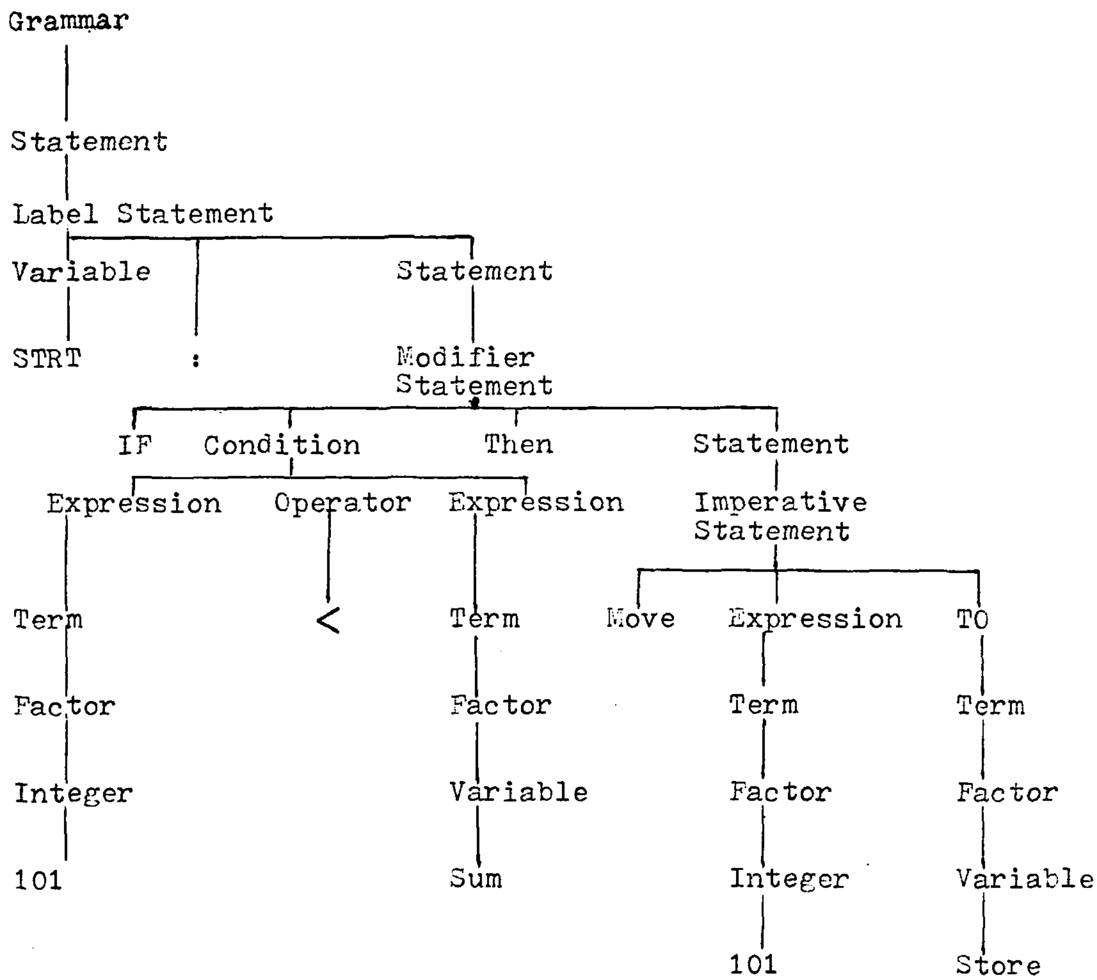

Figure 3. Syntax tree for sample sentence.

The process for parsing a sentence is through a father-son own-disown procedure. Each branch point is called a father and each branch is called a son. The father will adopt the first son (the first branch will be attempted) to see if he is the correct one. If not he will disown him and adopt the next one to see if he is valid. This procedure is continued through several levels if necessary until the last son is either adopted or rejected.

There are several other approaches for a parser design. One of the less frequently used is known as the bottom up technique. With this approach, the parse begins with the string itself and attempts to reduce it to the distinguished symbol. Using the example of Figure 3, the tree would be turned upside down to accomplish the parse. As stated, this is used less often than a top down because although structured after similar concepts in reverse, the bottom up is more difficult to define in a program table and it is more difficult to keep track of the parser's position in a table if one branch is tried and fails and another branch must be attempted.

Another method, also less frequently used is a precedence parse. This method is extremely difficult to use since the language must be defined such that each element of a statement must fall in a set order of precedence in relation to all other elements. The difficulty of programming almost precludes the use of this in complex languages.

The parser for AU78 as previously stated is a top-down parser. As tokens are passed from the scanner, the parser begins at the top level of the grammer and begins a search down a path to see if the

terminal symbol can be identified. If it cannot, the parser backs up to the previous division point (branch path) and attempts another path. This will continue until the terminal is identified or if one is not identified, then an error is indicated. Although this sounds like it would be a long, time consuming process, generally if the terminal is in error, this can be determined very rapidly with properly constructed grammer tables.

In AU78, this search process was accomplished against a grammer table loaded into core which contained all the possible paths a sentence could follow. As each path was searched, the point of departure was stored in a stack type approach, although the actual stack was not used because it was required for other procedures. If a path failed, the last entry on the stack would be recalled and the search would continue from that point down an alternative path until all paths were exhausted. It is interesting to note that in some complex sentences, over twenty-five stack positions had to be stored at one time to complete the sentence.

The parser also handles errors associated with the syntax of a sentence. It is here that an error such as the one described in Chapter II would be recognized. The way this works is that the parser expects certain elements to follow each other. For example, in a MOVE-TO statement, the parser expects a variable or integer to follow the MOVE which in turn is followed by the TO which must be followed by a variable name. If this exact sequence is not followed an error condition will result. In another example, the WRITE verb

allows either a variable or string to follow. It checks for a variable first and if one is not found, it will check for a string. If neither is found then there is an error in the WRITE statement. The method used for checking the constructs of a sentence are the tokens passed from the scanner with the class value, indexes and actual values. If the sentence is being built correctly the semantic routines are called for code generation. Figure 4 portrays a flow of parser procedures.

Symbol and String Tables

The AU78 semantic routines involve a two pass approach. This method was necessary because AU78 allows variables to be defined anywhere in a program and the first pass is used to build a symbol table and the second pass to append the symbol table values onto the generated object codes. In building a symbol table, each time a variable is encountered, the table is searched and compared to the variable value. If a match is found, the table remains unchanged. Upon completion of the first pass, the table elements are incorporated into the code already generated from source statements. This is accomplished by generating WORD statements which reserve areas for the variables in the object code.

Strings are also handled in a similar manner. Because a programmer could desire to write the same string in various places in a program, there is an optimizing feature which will store values in a table and each time a string is referenced it will compare it to determine if two are exactly equal. If they are, the string

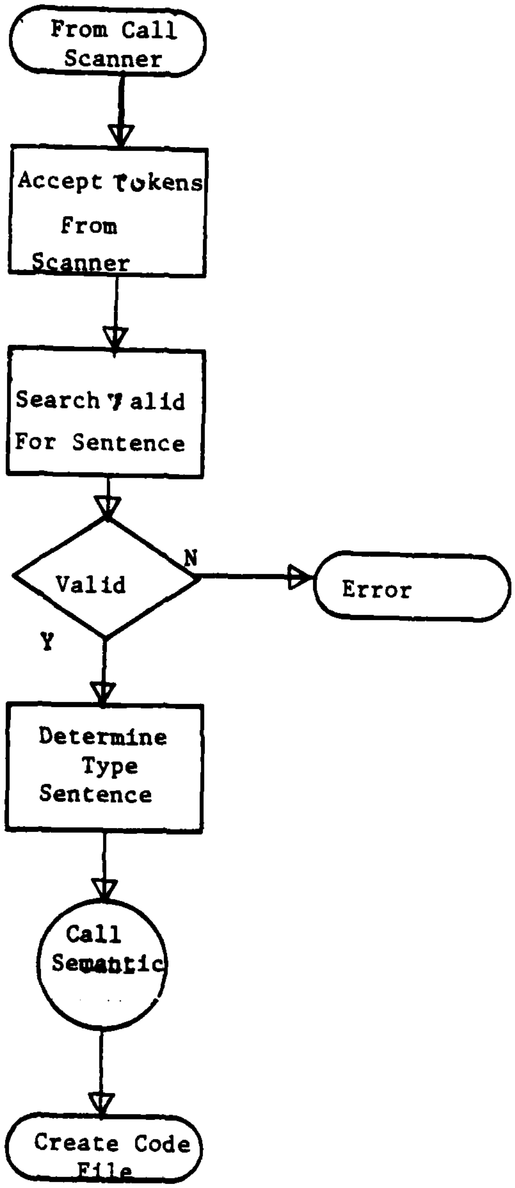

Figure 4. Flowchart of parser.

will be stored only once. At the beginning of the second pass, the string values and their lengths will be incorporated into the object code through the use of internally generated address labels which point to each string.

IV. SEMANTIC ROUTINES/CODE GENERATION

The semantic routines and code generation procedures are the parts of the cross-compiler which put the input source statements into internal formats and then generate object code. The semantic routines are responsible for creating symbol tables containing variable and/or labels and for putting the source statements passed on by the parser into the internal format such as Polish notation or quadruples. The code generation procedures accomplish exactly what the title implies, code generation. This is the most detailed and complicated part of the cross-compiler. However, it is also probably the best understood. These procedures take the created internal form and produce code for the sentences.[15] In the AU78 compiler, the two procedures have not been separated because they are imbedded within each other. For many straight-forward sentences such as READ or WRITE, the information passed from the parser is not changed. However for COMPUTE and IF-THEN-ELSE statements quadruples are created by the semantic routines. Consequently, in AU78 many statements have code generated without really being changed by the semantic routines; only the symbol table is created or updated. By interleaving the two routines wherever possible, AU78 was designed to provide a more efficient code generation process.

AU78 created code for each routine by using a subroutine approach for each type sentence. The remainder of this chapter will describe the process and considerations involved in developing code for each type statement. Figure 4 depicts the flow of the semantic routines/code generation procedures as they are called from the parser.

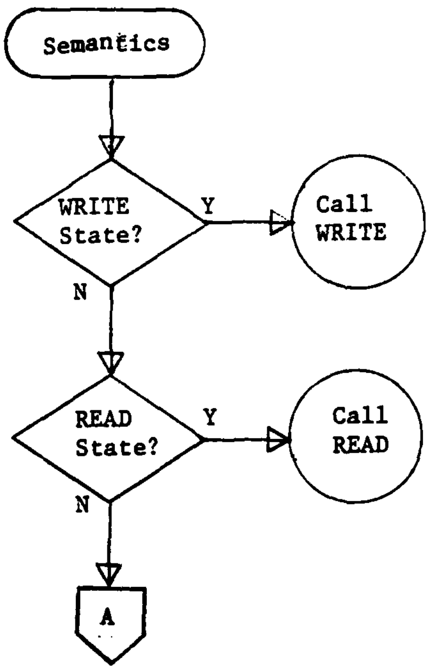

Figure 5. Semantic routine.

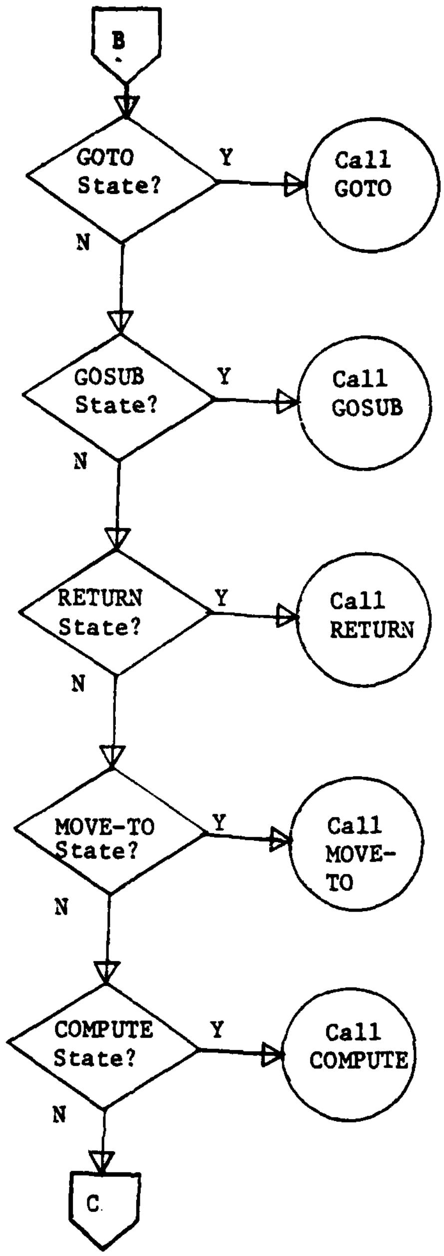

Figure 5. Semantic routine (Continued).

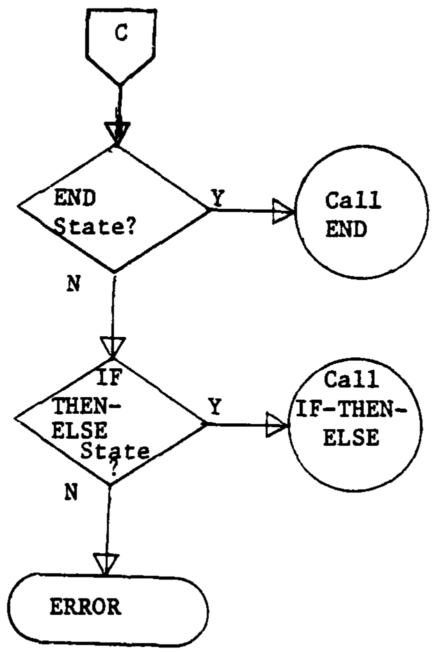

Figure 5. Semantic routine (Continued).

WRITE Statement

The WRITE statement code generation involves several considerations when producing the code. The WRITE can involve both the printing of variable values and the printing of strings. The printing of the string is relatively straightforward in that the output area is loaded with the length of the string and code is generated to print the string a character at a time, while the counter is decremented until the entire string is printed. In both cases, the code to check the status of the output device and to proceed when it is available is generated for the WRITE statement. Additionally, code to convert data to Binary Coded Decimal (BCD) is accomplished. A further capability of the WRITE statement is that the code is generated only once. All subsequent WRITE's call the subroutine from the first WRITE. The output variable is passed to the subroutine for printing as is a string and its length. By following this approach, much code is saved and a more optimal program is developed. This is accomplished by the setting of switches in the compiler to indicate if previous WRITE's have occurred. Figure 6 portrays a flowchart of the WRITE code generation which is called from the parser.

READ Statement

The READ statement allows the programmer to read an input value. When the READ variable is used, the variable area will receive the result of the input read procedure. The data is read into a buffer area and transferred to the variable area. As in the WRITE statement, code is generated which checks the status of the input device and

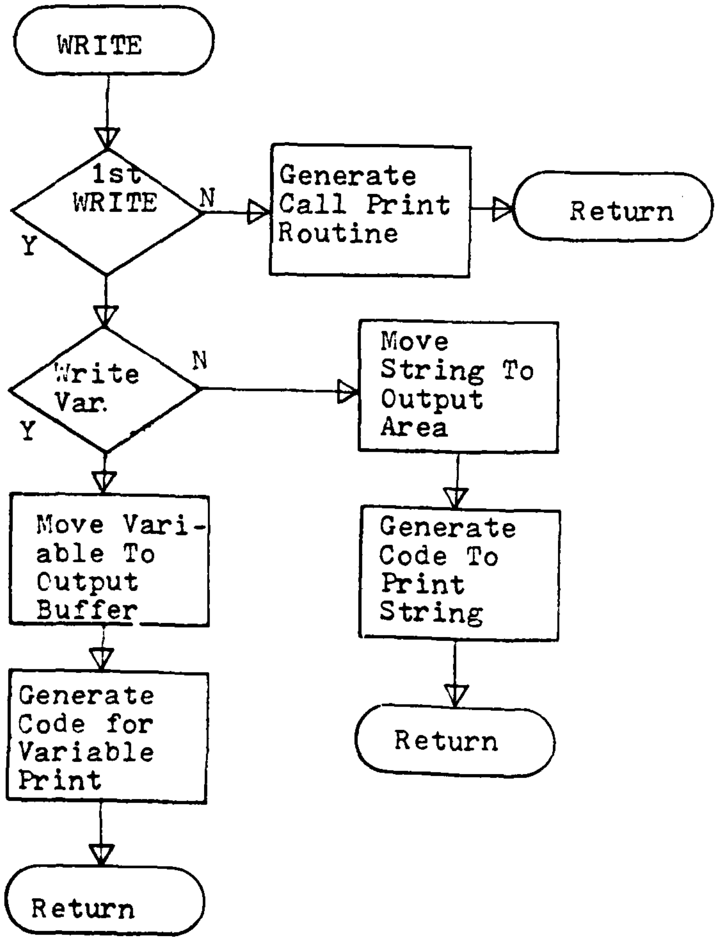

Figure 6. Write statement.

proceeds when the device is ready. All conversions from input BCD to internally usable hexidecimal are accomplished in the generated code. Also, code is generated for a READ only once. If additional READ's are used a subroutine call will use the code generated for the first READ and all variables will be passed to it. Figure 7 portrays a flowchart of the READ functions.

GOTO/GOSUB Statement

The GOTO and GOSUB statement is one of the easier statements for which to generate code. The GOTO and GOSUB must be followed by a valid label address. To generate code for the statement requires only an unconditional branch (JMP) in the case of a GOTO or subroutine call (CALL) in the case of a GOSUB to a labeled address. Even in assembly language, this requires only one instruction. Figures 8 and 9 show the flow of GOTO and GOSUB respectively.

RETURN Statement

For a subroutine to work effectively, a RETURN must be coded. If the RETURN is not included, when there is a call to a subroutine the processing will not return to the statement following the call statement, but will continue with the statement following the end of the subroutine. This could cause unpredictable results for the program execution. A RETURN statement can be labeled so that a direct exit from a subroutine is possible. Code generation for a RETURN also requires only one statement. Figure 10 depicts the flow for code generation for a RETURN statement.

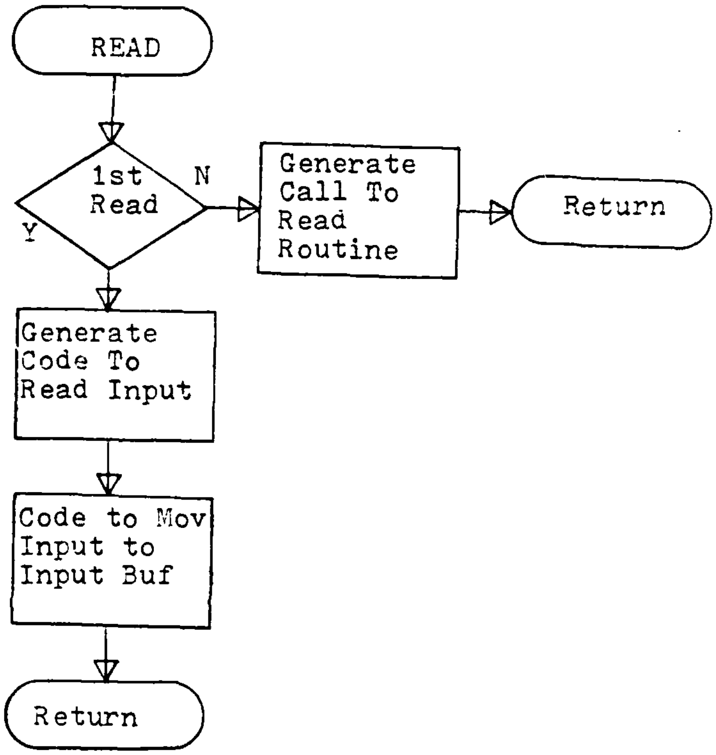

Figure 7. Read statement.

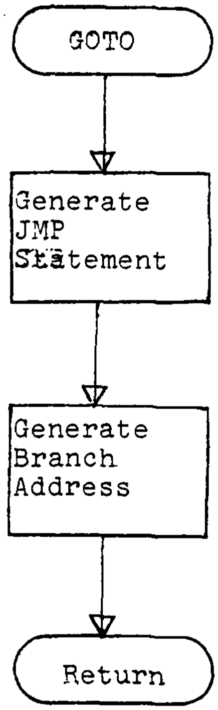

Figure 8. GOTO statement.

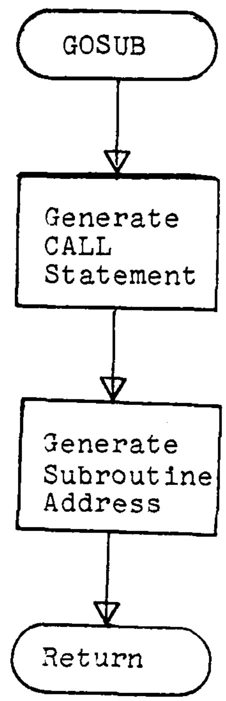

Figure 9. GOSUB statement.

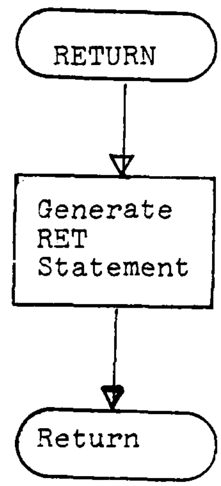

Figure 10. Return statement.

MOVE-TO Statement

The MOVE-TO statement allows for the internal transfer of data or the initialization of a variable. The data following the MOVE can either be a constant or variable but the receiving statement must be a variable. To generate code for either condition, the object code must be set up to handle either variable values or constants. To accomplish this the cross-compiler will determine if the value is a variable or constant and will generate specific code for each which will load the D and E registers with the constant value or variable value. This data is held until the TO condition is processed. The variable address following the TO is loaded into the H and L registers and a move statement is used to transfer the data to the address loaded in the register. Again the data in the sending field is not changed. Figure 11 is a flow description of the MOVE-TO statement.

COMPUTE Statement

The COMPUTE statement is the most difficult statement for which to generate code in the compiler. The COMPUTE allows for the computation of various equations involving both constants and variables and includes parenthesis to allow for greater depth. This presents a significant problem in being able to handle all the various combinations of conditions. Each addition, subtraction, multiplication, and division routine has to be uniquely written. However, for all the routines, a common procedure was established to load the registers with the two elements being computed at that time. In all routines, the same registers are used, making it possible for this to be done in a single subroutine. Figure 12 depicts the COMPUTE flow.

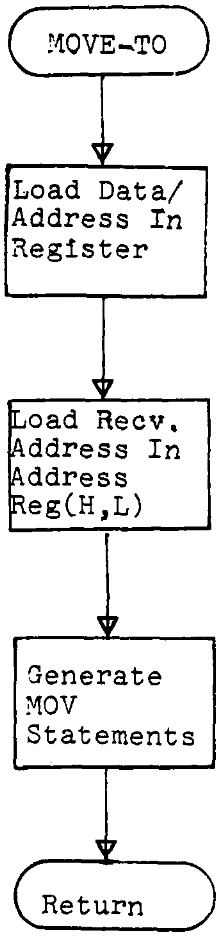

Figure 11. Move-To statement.

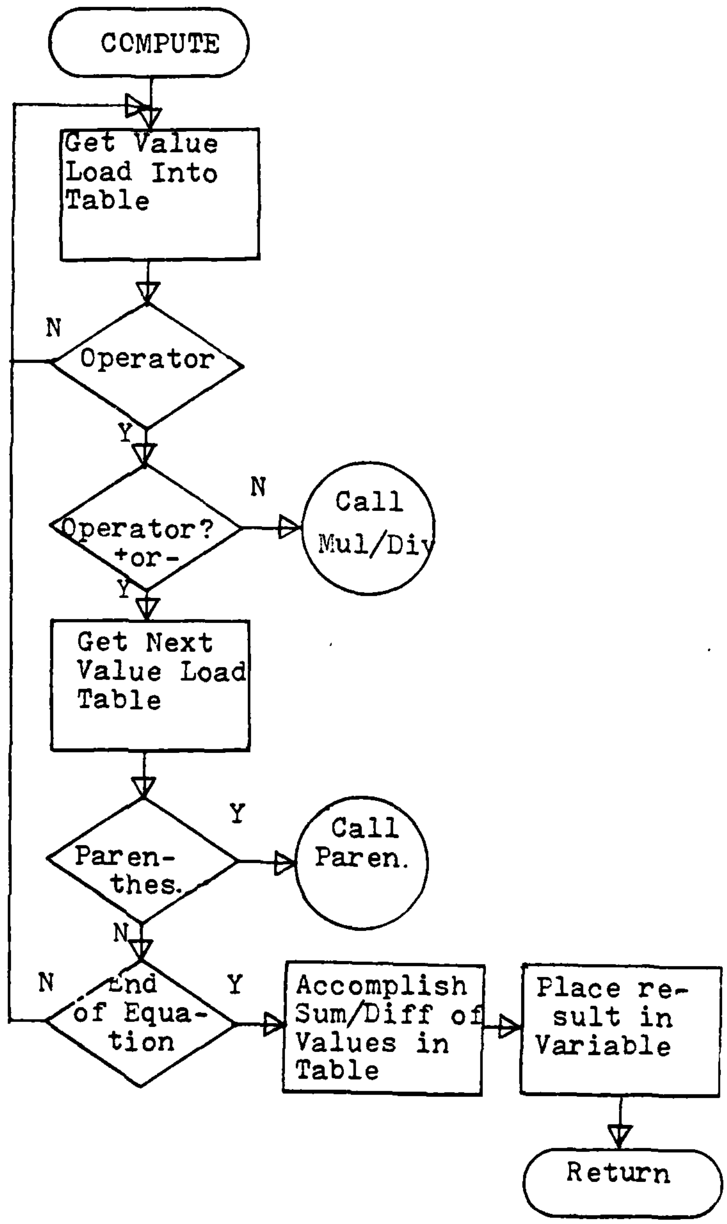

Figure 12. Compute statement.

The addition procedure was straightforward. Once the registers are loaded, a double add (DAD) is generated to accomplish the 16-bit addition with the results remaining in the D and E registers. For subtraction, the process is not quite as simple, since there is no double subtraction verb available in the assembly language. Therefore to accomplish 16-bit subtraction, the complement of the subtrahend must be taken and then a double add is preformed.

The multiplication procedure is more complex in design than either the addition or subtraction. To accomplish the multiplication, a procedure using a 16-bit right shift of the result and a right shift of the multiplier is performed. Each time the low order bit of the multplier is equal to one, the multiplier is added to the shifted high order byte of the result field. The procedure uses the B and C registers to hold the result, the C register initially holds the multiplier, the D register holds the multiplicand and the E register serves as a counter. This entire procedure is generated once in a program and if used again will be called as a subroutine and the registers loaded with the necessary values. Figure 13 depicts the flowchart for the multiplication procedure.

The division routine is less complex than the multiplication. It simply involves a series of subtractions which are accomplished by complementing the divisor and performing a double add (DAD) with the dividend. This procedure is followed until the dividend is zero or a remainder is found. The D and E registers hold the divisor, the H and L registers contain the dividend and the B and C registers contain the result. Again like the multiplication, the division

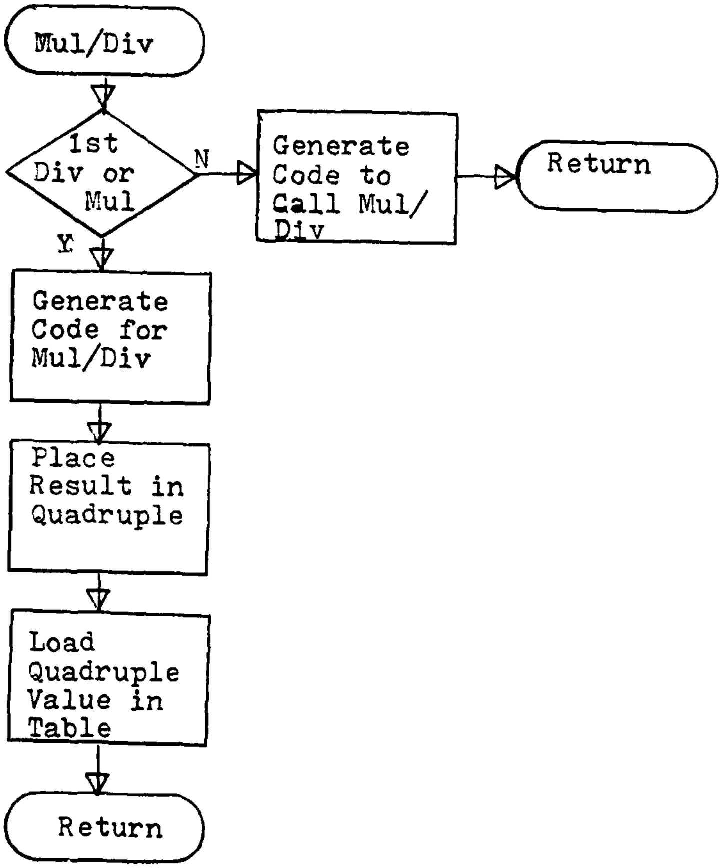

Figure 13. Multiply/Division subroutine.

procedure is written only once and subsequent uses are performed as a subroutine. Figure 13 is a flowchart of the division procedure.

To handle equations involving some or all of the possible operations, the compiler was designed to use a modified quadruple approach. In using quadruples, an equation is usually broken down into the two elements to be operated on, the operator and a result field in the general form value-value-result-operator.[16] The AU78 cross-compiler modified this approach somewhat and a table is used. Each element of the equation is loaded into the table as it is read. When an operator is read, it is checked to determine if it is a multiplication or division operator. If it is, a switch is set and after the next value is read, the operation is performed by calling either the multiplication or division subroutine. Upon completion of the operation, the results are stored in a unique result variable generated by the compiler. This result replaces the two values and the operator in the table and further reading of the equation proceeds. This will continue for all multplication and division problems while all additions or subtraction operations remain in the table. Figure 12 depicts the flow of the total operation.

While processing an equation, if a left parenthesis is encountered, the compiler will begin working with a new table following the same rules as for the first. In this table, all elements read after the parenthesis will be stored, and the multiplications and divisions performed until a right parenthesis is encountered; indicating the end of that computation. The additions and subtractions in the

second table will then be computed and the result for the entire operation enclosed in parenthesis will be stored in another unique compiler generated field. This result will be placed in the original table as a multiplication to be performed with the last entry in the table. Figure 14 depicts the flow of the parenthesis procedure. At the end of the equation, the cross-compiler will search the table, performing all additions and subtractions until the table is clear. The final result of the computation is placed in a specific cross-compiler generated result field which is then moved to the variable address indicated by the COMPUTE statement. The same tables are used for each computation, so only limited space is required to store the tables. The result is a fast, effective method for accomplishing computations.

END Statement

The END statement simply signifies to the compiler that this is the logical end of the program and the compiler will then generate an END verb. Figure 15 depicts the END condition.

IF-THEN-ELSE Statement

The IF-THEN-ELSE statement is another statement for which it is especially difficult and complex to generate code. In AU78, the conditional comparison allows either equations, variables, and/or constants to be compared. Consequently, the problem is quite complex in determining which type condition is being compared to which other condition. If equations are compared to each other or even to a variable or constant, the equation must first be computed and the

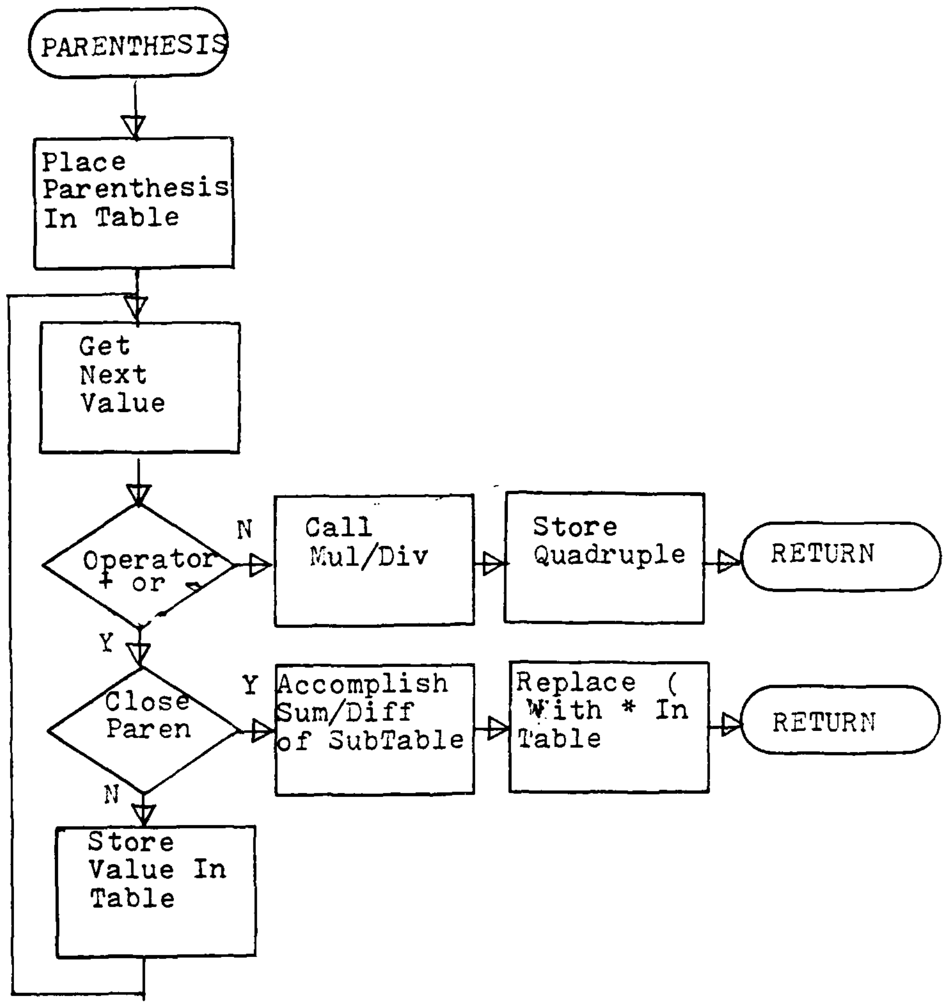

Figure 14. Parenthesis subroutine.

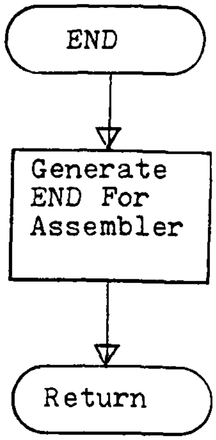

Figure 15. End statement.

result saved and then compared to the second value. When equations are being compared, the code for the equation will first be generated just as stated for the COMPUTE statement. If previous arithmetic subroutines have been generated which can satisfy the conditional equation, then only a subroutine call will be used and the entire code will not be recreated. The results of an equation will be stored long enough for the comparison to be accomplished after which the results are unavailable to the programmer. If it is necessary to have the results, then a COMPUTE statement should be performed and the results of it used in the comparison.

The method used for code generation for the condition is straightforward. The first value to be compared is stored in a compiler generated data field. The next value is then loaded into the D and E registers and based on the type comparison, code is generated which will compare the stored value to the register pair. Since this comparison is accomplished using the 8-bit accumulator, it must take place a single register at a time. The comparisons are made using the compare (CMP) verb and one of the jump verbs.

Since the next statement to be processed is based on the result of the comparison, these statements must be labeled so the program can jump to the correct one. To accomplish this, the cross-compiler generates address labels which are appended to the sentence following the THEN and the ELSE, or the next logical statement if the ELSE is not used. By using these labels, the comparison and resultant branch will cause a jump to the proper sentence. For each time an

IF-THEN-ELSE sentence is used, unique labels must be generated. This is accomplished by using a counter and appending the count to the end of a three character label, thereby insuring a unique label each time such as THN1, THN2, ELS1, and ELS2, Figure 16 shows the flow of the IF-THEN-ELSE statement.

As outlined above, the code generation process involves many intricacies not readily apparent until the detail design is underway. Also, the peculiarities of the language dictate to a great degree the procedures required to efficiently create code. However, the most important consideration is to generate efficient code and wherever possible duplicate code if it can be used by more than one statement. Code for high level languages can never be as efficient as that created directly in assembly language. But if care is taken and the design of the compiler well thought out, the differences in efficiency can become insignificant.

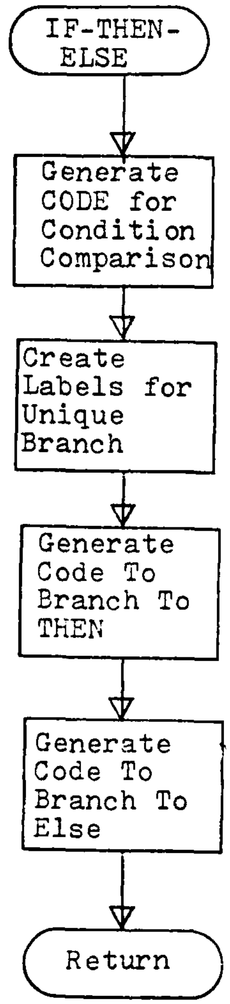

Figure 16. IF-THEN-ELSE statement.

V. SAMPLE AU78 PROGRAM AND LISTING

To demonstrate the capabilities of AU78 and the operation of the program on line, a sample program was developed and entered through a teletype keyboard to the I8080. The program accepts two input numbers and compares them to determine if the first is greater than or equal to the second. If the condition is met then the larger number is printed. If the condition is not met, then an equation is evaluated and the result printed. The main purpose of this program was to utilize each type statement in the AU78 language and to demonstrate the corresponding object code. Consequently, the flow of the program will produce at least one or two of each type statement. The source listing is found in Table 4. The input and output of the program on the teletype is found in Table 5. The object code generated by the cross-compiler is listed in Table 6. The symbol table is produced first followed by the actual operational code. The code generated for each source statement can be determined by tracing through the object code.

Table 4. Program source listing.

```
0100      GOSUB INP;
0110      MOVE INPUT TO NBR1;
0120      GOSUB INP;
0130      MOVE INPUT TO NBR2;
0140      IF NBR1>= NBR2 THEN GOTO GTR ELSE GOTO COM;
0150 GTR: MOVE NBR1 TO OUTPUT;
0160      GOTO OTP;
0170 COM: COMPUTE NBR1/2+10(NBR2*5-20)=OUTPUT;
0180 OTP: WRITE 'RESULT IS EQUAL TO ';
0190      WRITE OUTPUT;
0200      END;
0210 INP: READ INPUT;
0220      RETURN STOP
```

Table 5. Program operation.

```
8Ø8Ø V3.Ø

.G377Ø          ;Initialization procedure as defined in
                 Users Manual

8Ø8Ø V3.Ø

.G131           ;Start address of procedure

ØØØ8ØØØØA       ;Input

RESULT IS EQUAL TO 19  ;Output

8Ø8Ø V3.Ø       ;Ready
```

Table 6. Program generated object code.

```
>>>>>>>>>>SYMBOL TABLE<<<<<<<<<<

RESL9    0103
RESL8    0105
RESL7    0107
RESL6    0109
RESL5    010B
RESL4    010D
RESL3    010F
RESL2    0111
RESL1    0113
STG1     0115
OUTPUT   0128
NBR2     012A
NBR1     012C
INPUT    012E
SAV      0130
TTYST    0001
TTYIO    0000
INP      02EB
THN1     018D
ELS1     0193
GTR      0197
NXT2     0196
COM      01A8
OTP      026C
DIV      01B7
NUL5     01CF
DIVDE    01C3
FNDIV    01CD
MULT     01E5
NUL4     01FC
MULTO    01EA
DONE     01FB
MULT1    01F6
SUBT     0212
NUL3     0232
CARRY    021D
SECND    034B
DADD     0226
PRINT    0277
NUL6     0290
TTYOUT   0283
ENDPT    028F
TTYO     029B
NUL1     02E7
LTR1     02B3
TTO      02B5
TT02     02B8
LTR2     02DE
OT       02E0
CHK      02CC
OTRT     02E6
TTYIN    02F1
NUL2     0366
TTYSO    0300
```

Table 6. Program generated object code (continued).

```
NUMIN     031A
LTRIN     031C
SEC       033A
ASCII     032B
INXH      035B

\END OF PASS ONE



                                ORG  0100
0100 00                         BYTE  00
0101 00                         BYTE  00
0102 00                         BYTE  00
0103 00 00             RESL9:   WORD  00
0105 00 00             RESL8:   WORD  00
0107 00 00             RESL7:   WORD  00
0109 00 00             RESL6:   WORD  00
010B 00 00             RESL5:   WORD  00
010D 00 00             RESL4:   WORD  00
010F 00 00             RESL3:   WORD  00
0111 00 00             RESL2:   WORD  00
0113 00 00             RESL1:   WORD  00
0115 52                STG1: TXT  'RESULT IS EQUAL TO '
0116 45 53 55 4C 54 20 49 53 20 45 51 55 41 4C 20 54
0126 4F 20
0128 00 00             OUTPUT: WORD  00
012A 00 00             NBR2: WORD  00
012C 00 00             NBR1: WORD  00
012E 00 00             INPUT: WORD  00
0130 00                 SAV: BYTE 00
                       TTYST: EQU  01
                       TTYIO: EQU  00
0131 CD EB 02                 CALL INP
0134 21 2E 01                 LXI  H,INPUT
0137 5E                       MOV  E,M
0138 23                       INX  H
0139 56                       MOV  D,M
013A 21 2C 01                 LXI  H,NBR1
013D 7B                       MOV  A,E
013E 77                       MOV  M,A
013F 23                       INX  H
0140 7A                       MOV  A,D
0141 77                       MOV  M,A
0142 CD EB 02                 CALL INP
0145 21 2E 01                 LXI  H,INPUT
0148 5E                       MOV  E,M
0149 23                       INX  H
014A 56                       MOV  D,M
014B 21 2A 01                 LXI  H,NBR2
014E 7B                       MOV  A,E
014F 77                       MOV  M,A
0150 23                       INX  H
0151 7A                       MOV  A,D
```

Table 6. Program generated object code (continued).

```
0152 77                 MOV  M,A
0153 21 2C 01           LXI  H,NBR1
0156 5E                 MOV  E,M
0157 23                 INX  H
0158 56                 MOV  D,M
0159 21 13 01           LXI  H,RESL1
015C 73                 MOV  M,E
015D 23                 INX  H
015E 72                 MOV M,D
015F 21 13 01           LXI H,RESL1
0162 7E                 MOV  A,M
0163 5F                 MOV  E,A
0164 23                 INX  H
0165 7E                 MOV  A,M
0166 57                 MOV  D,A
0167 D5                 PUSH  D
0168 21 2A 01           LXI  H,NBR2
016B 5E                 MOV  E,M
016C 23                 INX  H
016D 56                 MOV  D,M
016E 21 13 01           LXI  H,RESL1
0171 73                 MOV  M,E
0172 23                 INX  H
0173 72                 MOV M,D
0174 D1                 POP  D
0175 21 13 01           LXI  H,RESL1
0178 23                 INX  H
0179 7E                 MOV  A,M
017A BA                 CMP  D
017B DA 8D 01           JC    THN1
017E C2 93 01           JNZ  ELS1
0181 2B                 DCX  H
0182 7E                 MOV  A,M
0183 BB                 CMP  E
0184 CA 8D 01           JZ   THN1
0187 DA 8D 01           JC   THN1
018A C3 93 01           JMP  ELS1
018D C3 97 01    THN1:  JMP  GTR
0190 C3 96 01           JMP  NXT2
0193 C3 A8 01    ELS1:  JMP  COM
0196 7F          NXT2:  MOV  A,A
0197 21 2C 01    GTR:   LXI  H,NBR1
019A 5E                 MOV  E,M
019B 23                 INX  H
019C 56                 MOV  D,M
019D 21 28 01           LXI  H,OUTPUT
01A0 7B                 MOV  A,E
01A1 77                 MOV  M,A
01A2 23                 INX  H
01A3 7A                 MOV  A,D
01A4 77                 MOV  M,A
01A5 C3 6C 02           JMP  OTP
01A8 11 02 00    COM:   LXI  D,2
01AB 21 2C 01           LXI  H,NBR1
01AE 4E                 MOV  C,M
01AF 23                 INX  H
01B0 46                 MOV  B,M
```

Table 6. Program generated object code (continued).

```
01B1 CD B7 01          CALL  DIV
01B4 C3 CF 01           JMP  NUL5
01B7 7A            DIV: MOV  A,D
01B8 2F                 CMA
01B9 57                 MOV  D,A
01BA 7B                 MOV  A,E
01BB 2F                 CMA
01BC 5F                 MOV  E,A
01BD 13                 INX  D
01BE 69                 MOV  L,C
01BF 60                 MOV  H,B
01C0 01 00 00           LXI  B,0
01C3 19          DIVDE: DAD  D
01C4 7C                 MOV  A,H
01C5 17                 RAL
01C6 DA CD 01           JC   FNDIV
01C9 03                 INX  B
01CA C3 C3 01           JMP  DIVDE
01CD AF          FNDIV: XRA  A
01CE C9                 RET
01CF 7F           NUL5: MOV  A,A
01D0 21 13 01           LXI  H,RESL1
01D3 71                 MOV  M,C
01D4 23                 INX  H
01D5 70                 MOV  M,B
01D6 11 05 00           LXI  D,5
01D9 21 2A 01           LXI  H,NBR2
01DC 4E                 MOV  C,M
01DD 23                 INX  H
01DE 46                 MOV  B,M
01DF CD E5 01           CALL MULT
01E2 C3 FC 01           JMP  NUL4
01E5 06 00        MULT: MVI  B,0
01E7 53                 MOV  D,E
01E8 1E 09              MVI  E,9
01EA 79          MULTO: MOV  A,C
01EB 1F                 RAR
01EC 4F                 MOV  C,A
01ED 1D                 DCR  E
01EE CA FB 01           JZ   DONE
01F1 78                 MOV  A,B
01F2 D2 F6 01           JNC  MULT1
01F5 82                 ADD  D
01F6 1F          MULT1: RAR
01F7 47                 MOV  B,A
01F8 C3 EA 01           JMP  MULTO
01FB C9           DONE: RET
01FC 7F           NUL4: MOV  A,A
01FD 21 11 01           LXI  H,RESL2
0200 71                 MOV  M,C
0201 23                 INX  H
0202 70                 MOV  M,B
0203 11 14 00           LXI  D,20
0206 21 11 01           LXI  H,RESL2
0209 4E                 MOV  C,M
020A 23                 INX  H
020B 46                 MOV  B,M
```

Table 6. Program generated object code (continued).

```
020C CD 12 02          CALL  SUBT
020F C3 32 02           JMP  NUL3
0212 7B         SUBT: MOV   A,E
0213 2F               CMA
0214 C6 01            ADI   1
0216 5F               MOV   E,A
0217 DA 1D 02         JC    CARRY
021A C3 4B 03         JMP   SECND
021D 7A         CARRY: MOV  A,D
021E 2F               CMA
021F C6 01            ADI   1
0221 57               MOV   D,A
0222 C3 28 02         JMP   DADD
0225 7A         SECND: MOV  A,D
0226 2F               CMA
0227 57               MOV   D,A
0228 EB          DADD: XCHG
0229 09               DAD   B
022A EB               XCHG
022B 21 0F 01         LXI   H,RESL3
022E 73               MOV   M,E
022F 23               INX   H
0230 72               MOV   M,D
0231 C9                RET
0232 7F         NUL3: MOV   A,A
0233 21 0F 01         LXI   H,RESL3
0236 5E               MOV   E,M
0237 23               INX   H
0238 56               MOV   D,M
0239 01 0A 00         LXI   B,10
023C CD E5 01         CALL  MULT
023F 21 0F 01         LXI   H,RESL3
0242 71               MOV   M,C
0243 23               INX   H
0244 70               MOV   M,B
0245 21 0F 01         LXI   H,RESL3
0248 5E               MOV   E,M
0249 23               INX   H
024A 56               MOV   D,M
024B 21 13 01         LXI   H,RESL1
024E 4E               MOV   C,M
024F 23               INX   H
0250 46               MOV   B,M
0251 EB               XCHG
0252 09               DAD   B
0253 EB               XCHG
0254 21 0D 01         LXI   H,RESL4
0257 73               MOV   M,E
0258 23               INX   H
0259 72               MOV   M,D
025A 21 13 01         LXI   H,RESL1
025D 73               MOV   M,E
025E 23               INX   H
025F 72               MOV M,D
0260 21 13 01         LXI   H,RESL1
0263 56               MOV   D,M
0264 23               INX   H
```

Table 6. Program generated object code (continued).

```
0265 5E               MOV  E,M
0266 21 28 01         LXI  H,OUTPUT
0269 72               MOV  M,D
026A 23               INX  H
026B 73               MOV  M,E
026C 21 15 01    OTP: LXI  H,STG1
026F 06 13            MVI  B,19
0271 CD 77 02         CALL PRINT
0274 C3 90 02          JMP  NUL6
0277 4E         PRINT: MOV  C,M
0278 CD 83 02         CALL  TTYOUT
027B 23               INX  H
027C 05               DCR  B
027D C2 77 02         JNZ  PRINT
0280 C3 8F 02         JMP  ENDPT
0283 DB 01     TTYOUT: IN    TTYST
0285 E6 04            ANI  04
0287 C2 83 02         JNZ  TTYOUT
028A 79               MOV  A,C
028B 2F               CMA
028C D3 00            OUT  00
028E C9               RET
028F C9        ENDPT: RET
0290 7F         NUL6: MOV  A,A
0291 21 28 01         LXI  H,OUTPUT
0294 23               INX  H
0295 CD 9B 02         CALL  TTYO
0298 C3 E7 02         JMP  NUL1
029B DB 01      TTYO:  IN   TTYST
029D E6 04            ANI  04
029F C2 9B 02         JNZ  TTYO
02A2 7E               MOV  A,M
02A3 E6 F0            ANI  240
02A5 0F               RRC
02A6 0F               RRC
02A7 0F               RRC
02A8 0F               RRC
02A9 FE 0A            CPI  0A
02AB F2 B3 02         JP   LTR1
02AE C6 30            ADI  030
02B0 C3 B5 02         JMP  TTO
02B3 C6 37      LTR1: ADI  037
02B5 2F         TTO: CMA
02B6 D3 00            OUT  TTYIO
02B8 DB 01      TT02: IN   TTYST
02BA E6 04            ANI  04
02BC C2 B8 02         JNZ  TT02
02BF 7E               MOV  A,M
02C0 E6 0F            ANI  0F
02C2 FE 0A            CPI  0A
02C4 F2 DE 02         JP   LTR2
02C7 C6 30            ADI  030
02C9 C3 E0 02         JMP  OT
02CC 21 30 01    CHK: LXI  H,SAV
02CF 7E               MOV  A,M
02D0 FE 01            CPI  1
02D2 CA E6 02          JZ  OTRT
```

Table 6. Program generated object code (continued).

```
02D5 C6 01             ADI   1
02D7 77                MOV   M,A
02D8 21 28 01          LXI   H,OUTPUT
02DB C3 9B 02          JMP   TTYO
02DE C6 37       LTR2: ADI   037
02E0 2F            OT: CMA
02E1 D3 00             OUT   TTYIO
02E3 C3 CC 02          JMP   CHK
02E6 C9          OTRT: RET
02E7 7F          NUL1: MOV   A,A
02E8 C3 00 38          JMP   03800
02EB CD F1 02     INP: CALL  TTYIN
02EE C3 66 03          JMP   NUL2
02F1 DB 01      TTYIN: IN    TTYST
02F3 E6 01             ANI   01
02F5 C2 F1 02          JNZ   TTYIN
02F8 DB 00             IN  TTYIO
02FA 2F                CMA
02FB E6 7F             ANI   07F
02FD 32 02 01          STA   0102
0300 DB 01      TTYSO: IN    TTYST
0302 E6 04             ANI   04
0304 C2 00 03          JNZ   TTYSO
0307 3A 02 01          LDA   0102
030A 2F                CMA
030B D3 00             OUT   TTYIO
030D 3A 02 01          LDA   0102
0310 FE 3A             CPI   03A
0312 DA 1A 03          JC   NUMIN
0315 D6 37             SUI   037
0317 C3 1C 03          JMP   LTRIN
031A D6 30      NUMIN: SUI   030
031C 32 02 01   LTRIN: STA   0102
031F 21 30 01          LXI   H,SAV
0322 7E                MOV   A,M
0323 FE 01             CPI   1
0325 CA 3A 03          JZ    SEC
0328 C6 01             ADI   1
032A 77                MOV   M,A
032B 21 02 01   ASCII: LXI   H,0102
032E 01 00 01          LXI   B,0100
0331 7E                MOV   A,M
0332 07                RLC
0333 07                RLC
0334 07                RLC
0335 07                RLC
0336 02                STAX B
0337 C3 F1 02          JMP   TTYIN
033A 21 02 01     SEC: LXI   H,0102
033D 7E                MOV   A,M
033E E6 0F             ANI   0F
0340 21 00 01          LXI   H,0100
0343 B6                ORA   M
0344 02                STAX B
0345 7B                MOV   A,E
0346 FE 01             CPI   1
0348 CA 5B 03          JZ    INXH
```

Table 6. Program generated object code (continued).

```
034B 1E 01        SECND: MVI  E,1
034D 21 30 01            LXI  H,SAV
0350 AF                  XRA  A
0351 77                  MOV  M,A
0352 0A                  LDAX B
0353 21 2E 01            LXI  H,INPUT
0356 23                  INX  H
0357 77                  MOV  M,A
0358 C3 F1 02            JMP  TTYIN
035B 21 2E 01     INXB:  LXI  H,INPUT
035E 0A                  LDAX B
035F 77                  MOV  M,A
0360 21 30 01            LXI  H,SAV
0363 AF                  XRA  A
0364 77                  MOV  M,A
0365 C9                  RET
0366 7F           NUL2:  MOV  A,A
0367 C9                  RET
0368 C3 00 38            JMP  03800
                         END

\END OF PASS TWO
```

VI. CONCLUSION

The design and development of the AU78 language and compiler has provided at least one significant advancement. It has opened a way to more efficient programming of an Intel 8080 based microcomputer using a simple high level language. Using the assembly language for the I8080 is tedious at best and the introduction of AU78 provides a basis for increasing a programmers capacity to fully utilize the I8080. It is realized AU78 has limitations. It was designed as a general purpose type language and as a result specific capabilities to accomplish a specific task may not be present. However, the basis for the inclusion of these capabilities is available through the modular design of the compiler. Additional capabilities could be added to perform a specific task by adding statements to the grammar and the grammar table and by creating a new subroutine.

Major achievements, as far as the author is concerned, are the tremendous experience gained in completing such a task and providing the capability to program the I8080 in a high level language using the time share capabilities of the RSTS/E operating system of a DEC PDP11/40 minicomputer. It is virtually impossible to fully understand the inner workings of a compiler without writing one. Some compilers take several persons at least a year or two to write and even then they are seldom complete or totally accurate. Most are continually being enhanced and modified because of the detailed problems

associated with accounting for every possible way a source language statement can be used. To better understand a portion of the problem involved in valuable knowledge.

One of the most important lessons learned from this effort is that to be most effective, a high level language for a small computer should be designed for a specific type application. To design one for many type uses requires trade-offs in capabilities which tend to weaken the overall language. For large machines, this is not as critical, as evidenced by PL/1. But where core is a significant consideration, special purpose languages are best. The AU78 requires 15K core on the PDP11/40, but had it been designed for a specific application, these requirements could have been reduced.

Another important lesson is that in designing a compiler, one should map out the capabilities it will have at the start and develop these first. Otherwise in the development, it is very easy to desire to add more capabilities. This in itself is not bad, but it tends to extend the estimated completion dates. In other words, establish the initial capabilities and enhance later.

Finally, it should be noted that many efforts to develop high level languages for microcomputers have already been undertaken. The results of all so far have not been totally satisfactory in respect to broad capabilities. However, many of these efforts are relatively recent and progress is being made in the design and efficiency of these languages. With additional work, the high level language future for microcomputers seems promising. The author sees AU78 as a small step in that direction.

REFERENCES

[1]B. A. Perrin, "High Level Languages and the Microprocessor," Electronic Engineering, May 1977, p. 65.

[2]Ibid.

[3]Ibid., p. 66

[4]Mark Alexander, "An Inside Look Into NIBL-Extended Tiny BASIC for the SC/MP," Interface Age, January 1977, p. 106.

[5]Yaohan Chu, Computer Organization and Microprogramming, (Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1972), p. 173.

[6]Lou Frenzel, "How to Choose a Microprocessor," BYTE, July 1978, p. 138.

[7]M. H. Hamza, ed., IEEE 1976 Mini-Mini and Microcomputers (IEE Computer Society, 1977), p.43.

[8]John Coach and Terry Hamm, "Semantic Structures for Efficient Code Generation on a Stack Machine", Computer, May 1977, p. 140.

[9]Perrin, "High Level Languages," p. 66.

[10]W. M. Kennan, J. J. Horning, and D. B. Wortman, A Compiler Generator (Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1970), p.43.

[11]Donald C. Simoneaux, High Level Language Compiling for User-Definable Architecture (Cameron Station, Alexandria, Virginia: Defense Documentation Center, 1975), p.24.

[12]John J. Donovan, Systems Programming (New York: McGraw-Hill Book Company, 1972), p. 266.

[13]David Gries, Compiler Construction for Digital Computers (New York: John Wiley and Sons, Inc., 1971), p. 65.

[14]Donovan, Systems Programming, p. 26[illegible]

[15] Gries, Compiler Construction, p. 5-6.

[16] Gries, Compiler Construction, p. 5-6.

APPENDIX A

USER'S MANUAL

AU78 USER'S MANUAL

1.0 General Procedures - The development of an AU78 language program must be accomplished through an on-line PDP11 remote terminal for compilation by a BASIC program with the results being stored in a user's disk file.

1.1 Log on Procedures - The user will log onto the PDP11 in the usual manner as determined by the local environment. Generally an account number and a password are required entries.

1.2 Run of Compiler - When the READY signal appears on the terminal the compiler can be initiated. At that point, enter the command RUN % AU78. When the program is loaded, a "?" will appear. The compiler is ready to accept source code.

2.0 Coding - Source code can be input in a generally free form format with only spaces required between entires.

2.1 Line Numbers - Line numbers are optional in AU78 and can be used if desired by the programmer. Since they are not required, branch conditions cannot be to line numbers and must be to labeled statements.

2.2 Labeled Statements - Any AU78 statement can be labeled. A label must begin with an alphabetic character and can be up to five characters long. The label must be followed by a colon before the program statement begins.

2.3 Sentences - A sentence is any valid AU78 statement. A semicolon should be placed at the end of each sentence to signify its completion.

2.4 Lines of Code - Upon completion of a sentence, a TAB character on the keyboard can be depressed if it is desired that the next sentence began on a new line. This is not necessary since the lines can continue without interruption.

2.5 Logical End of Program - The logical end of a program is signified by the insertion of an END statement.

2.6 Subroutines - Subroutines can physically be placed anywhere in a program except as the very first statement. A call to a subroutine will cause a branch to the subroutine address, perform the function and return to the next logical statement after the call. If the subroutines are to be grouped at the end of the program, the END statement must be entered before the subroutines are added.

2.7 Physical End of Program - When all coding is complete, a STOP statement followed by a carriage return should be entered to signify to the compiler the last statement has been entered and compilation is to begin.

3.0 Compilation and Assembly - If an error were detected during compilation, they should be corrected and rerun. Upon successful completion of a compilation, the assembly process is ready to run.

3.1 Assembly - To assemble the compiled code, enter RUN % ASMBLR. Then the following questions and their responses are used to run the assembler.

MICROCOMPUTER ? 8Ø8Ø

INPUT FILE ? COMP. DAT

OUTPUT FILE ? Enter either the name of a file or depress the carriage return for a paper tape file.

LISTING ? Enter /Q for the printer or any valid PDP11 RSTS entry.

3.2 Sign-Off - Upon completion of the assembly process, sign off the system with a BYE command.

4.0 Procedure for Intellec - 8 - The following actions are required to initialize the Intellec-8 for operation

-Turn on the Intellec - 8

-Turn teletype to "on line" position.

-Press "MEMORY ACCESS" (TOP HALF).

-Press "WAIT".

-Place zero in the switch register by depressing the lower half of each of 16 switches in the Switch register.

-Press "LOAD"

-Place "C8" (hexadecimal) in the eight switches in the right half of the switch register. The switch is in the one position if the upper half of the switch is pressed in.

-Press "DEP".

-Press "INCR".

-Place "ØØ" in the right half of the switch register.

-Press "DEP".

-Press "INCR".

-Place "38" (hexadecimal) in the right half of the switch register.

-Press "DEP".

-Press "RESET".

-Return the memory access switch to the "RESET" position.

-Return the wait switch to the "RESET" position.

-Message should then be printed on the teletype.

4.1 Procedure to load code - To load a paper tape in the Intellec - 8 the following steps should be followed.

-Turn on and bring up Intellec-8.

-Type "G377Ø" followed by a carriage return.

-Depress "Start" on paper tape reader.

-Program will return to monitor when load is complete.

4.2 Procedure to Execute program - After the tape is loaded, type GXXXX followed by a carriage return. The XXXX is the address of the starting location of the program in hexidecimal.

5.0 AU78 Statements - The statements acceptable by the AU78 compiler can be entered in a free form format. The only exception is that a space must be placed between key words and other entries.

5.1 READ variable - The READ statements accepts input data into the area set up by the variable. Data can be either processed in the variable area or moved to another area for processing. The status of the input device is automatically checked by the READ statement.

5.2 WRITE variable/string - The WRITE statement will write to the output device the information stored in the variable area or will print out the entire string which is enclosed by single quotes. The string capability allows the programmer to set up header and format information. The status of the output device is automatically checked by the WRITE statement.

5.3 MOVE variable/integer TO variable - The MOVE-TO statement allows for the transfer of data internally in the program. It can also be used to initialize a variable or set/reset a variable value. With the MOVE-TO, the value of the sending field remains unchanged.

5.4 GOTO Label - The GOTO statement accomplishes a transfer of control from the current location to the location of the label address. A GOTO should not be used to branch out of a subroutine since it could create execution problems later in the program processing.

5.5 GOSUB Label - The GOSUB statement transfers program control from the current statement to the address identified by the label. Upon completion of the subroutine, control is passed back to the sentence following the GOSUB. DATA can be passed to subroutines through variables. If the variables are changed in a subroutine, the new values will be available for use in the main program.

5.6 RETURN - The RETURN statement provides a means of return from a subroutine to the main body of the program. The first statement after the GOSUB statement will be executed immediately after the RETURN statement.

5.7 COMPUTE expression = variable - The COMPUTE statement provides a means of computing equations. The use

of parenthesis is permitted to provide greater flexibility and depth in equation development. All equations are evaluated in proper order with operations within parenthesis being accomplished first, followed by multiplication and division operations. Both variables and integers may be used in a typical equation. The results must be equated to a variable for further processing.

5.8 IF condition (equation/variable/integer - operation - equation/variable/integer) THEN statement ELSE statement - The IF-THEN-ELSE statement provides the capability to perform comparisons between values and then accomplish certain operations based on the result of the comparison.

5.8.1 IF condition - The IF condition allows several alternative bases for comparison. A full mathematical equation can be compared to another equation, variable or integer. Additionally, variables or integers can be compared with each other or with equations. It should be noted that if an equation is used as the basis for comparison, the results of the equation are not saved beyond the use of the statement.

5.8.2 THEN statement - The THEN statement will be executed if the IF condition is met. Unless the THEN is followed

by a branch statement, upon completion of the statement, control will pass to the statement following the IF-THEN-ELSE.

5.8.3 ELSE statement - The ELSE statement is optional. If used, the statement following the ELSE will be executed if the conditional comparison fails. Unless a branch statement is used, upon completion of the statement the program control will fall through to the next sentence. If the ELSE statement is not used, the conditional statement fails, control will pass to the next sentence.

5.9 END - The END statement identifies the logical end of the program.

5.10 STOP - The STOP statement is used to indicate the last statement in the program.

6.0 Key Words - The following is a list of key words which cannot be used except as stated above.

```
IF          RETURN
THEN        COMPUTE
ELSE        READ
GOTO        WRITE
GOSUB       END
            STOP
```

7.0 Reserved Words - The following is a list of reserved words that cannot be used in coding on AU78 program:

A	M	SP
B	OUT	THN (1-20)
C	PUSH	ELS (1-20)
D	POP	STG (1-20)
E	CALL	STOP
H	ORG	NUL (1-9)
I	EQU	DIVDE
STAY	SET	FNDIO
DADD	PSW	DIV
SECND	CARRY	SUBT
PRINT	TTYOUT	ENDPT
TTO	LTR1	TTO2
CHK	LTR2	OT
MULT	MULT1	NUM1N
LTRIN	MULTO	INXH
TTYIN	SEC NO	SEC
		TTYSO

8.0 Special Characters - With the exception of the special characters listed below, other special characters are not allowed except within strings. The following may be used with AU78 code.

*
/
+
-
;
>=
<=
=
(
)
'
<>

APPENDIX B

AU78 PROGRAM LISTING

```
1      DIM G$(419)\MAT READ G$
2      DATA 0,0,0
4      DATA 0,1,0,332
8      DATA 6,20,0,12
12     DATA 6,4,16,16
16     DATA 3,10,0,1
20     DATA 0,2,0,336
24     DATA 6,36,0,28
28     DATA 4,5,1,32
32     DATA 6,20,0,1
36     DATA 0,3,0,40
40     DATA 6,52,44,1
44     DATA 6,124,48,1
48     DATA 6,192,0,1
52     DATA 0,4,0,56
56     DATA 3,1,0,60
60     DATA 6,80,0,64
64     DATA 3,2,0,68
68     DATA 6,124,0,72
72     DATA 3,9,1,76
76     DATA 6,124,0,1
80     DATA 0,5,0,84
84     DATA 6,212,0,88
88     DATA 6,96,0,92
92     DATA 6,212,0,1
96     DATA 0,6,0,100
100    DATA 4,1,104,1
104    DATA 4,2,108,1
108    DATA 4,11,112,1
112    DATA 4,12,116,1
116    DATA 4,13,120,1
120    DATA 4,3,0,1
124    DATA 0,7,0,128
128    DATA 3,3,136,132
132    DATA 6,184,0,1
136    DATA 3,4,148,140
140    DATA 6,184,144,1
144    DATA 6,300,0,1
148    DATA 3,5,156,152
152    DATA 6,208,0,1
156    DATA 3,6,164,160
160    DATA 6,208,0,1
164    DATA 3,7,180,168
168    DATA 6,208,340,172
172    DATA 3,11,0,176
176    DATA 6,208,0,1
180    DATA 3,8,316,1
184    DATA 0,8,0,188
188    DATA 6,208,0,1
192    DATA 0,9,0,196
196    DATA 6,208,0,200
200    DATA 4,4,0,204
204    DATA 6,36,0,1
208    DATA 0,10,0,304
212    DATA 0,11,0,216
216    DATA 6,228,0,220
220    DATA 6,272,1,224
```

```
224     DATA 6,212,0,1
228     DATA 0,12,0,232
232     DATA 6,248,0,236
236     DATA 6,284,344,240
240     DATA 6,212,0,1
244     DATA 0,0,0,0
248     DATA 0,13,0,252
252     DATA 4,14,264,256
256     DATA 6,212,0,260
260     DATA 4,15,0,1
264     DATA 6,296,268,1
268     DATA 6,208,0,1
272     DATA 0,14,0,276
276     DATA 4,6,280,1
280     DATA 4,7,0,1
284     DATA 0,15,0,288
288     DATA 4,8,292,1
292     DATA 4,9,0,1
296     DATA 0,16,0,308
300     DATA 0,17,0,312
304     DATA 2,0,0,1
308     DATA 1,0,0,1
312     DATA 5,0,0,1
316     DATA 3,12,356,320
320     DATA 6,212,0,324
324     DATA 4,3,0,328
328     DATA 6,208,0,1
332     DATA 1,0,8,8
336     DATA 1,0,24,24
340     DATA 6,296,0,172
344     DATA 4,14,1,348
348     DATA 6,212,0,352
352     DATA 4,15,0,1
356     DATA 3,13,0,1
360     DATA 7,6,0,1
364     DATA 7,7,0,172
368     DATA 7,8,0,1
372     DATA 7,9,0,200
376     DATA 7,10,0,88
380     DATA 7,11,0,92
384     DATA 7,12,0,1
388     DATA 7,13,0,68
392     DATA 7,14,0,72
396     DATA 7,15,0,1
400     DATA 7,16,0,1
404     DATA 7,17,0,1
408     DATA 7,18,0,256
412     DATA 7,19,0,1
416     DATA 7,20,0,1
500     DIM KS(13)\MAT READ KS
520     DATA IF, THEN,READ,WRITE,GOTO,GOSUB,MOVE,RETURN,ELSE,STOP,TO
530 DIM AS(20)
540 A%=1\M1%=0
        \D2%=0
        \S2%=0
        \P1%=0
        \I1%=0
```

```
545 DIM D$(20)\D%=1\E1%=1\T1%=1
550 OPEN "COMP.DAT" FOR OUTPUT AS FILE 1%
560 OPEN "TEMP.TMP" FOR OUTPUT AS FILE 2%
570 DIM E2$(20)\E2%=1\S1%=0\R1%=1
1000 REM SCANNER
1010 DIM T$(17)\MAT READ T$
1020    DATA GRAMMER,LINE,STATEMENT,MODSTATE,CONDITION,OPERATOR,IMPST
GER,STRING
1030 T%=1
1040 DIM S%(30)
1050 H%=4\I%=0
1055 Y%=0\WS="      "
1060 INPUT LINE S$
1070 GOSUB 1280\P%=4\GOSUB 1090
1074 ZS="      END"\GOSUB 32000\CLOSE 2%
1075 OPEN "TEMP.TMP" FOR INPUT AS FILE 2%\GOSUB 20000
     \CLOSE 1%\CLOSE 2%
1080 STOP
1090 REM
1100 REM G%IS GRAMMER MATRIX
1110 REM T$ IS TRACE NAME VECTOR ,T%=1 FOR TRACE
1120 R%=0\O%=G%(P%)
1130 IF O%=0 THEN R%=1\IF T%=1 THEN PRINT TAB(X%*3);T$(G%(P%+1))
1140 IF (O%=1 OR O%=2 OR O%=5) AND C%=O% THEN R%=1:IF T%=1 THEN PRIN'
1150 IF (O%=3 OR O%=4) AND C%=O% AND V%=G%(P%+1) THEN R%=1\IF T%=1 TI
1155 GOSUB 32500
1160 IF O%>0 AND O%<6 AND R%=1 THEN GOSUB 1280
1170 IF O%=6 THEN X%=X%+1\S%(X%)=P%\P%=G%(P%+1)\GOTO 1120
1180 REM SEMANTIC DECODE HERH
1200 P%=G%(P%+2+R%)
1205 IF P%=95 THEN GOSUB 4700
1208 IF P%=72 AND C%=3 AND V%=9 THEN E6%=1
1209 IF P%=28 AND C%=4 AND V%=5 AND E6%=1 THEN GOSUB 6200\E6%=0
1210 IF P%=2 THEN X%=0\P%=0
1220 IF NOT (P%=0 OR P%=1) THEN GOTO 1120
1230 R%=P%\P%=S%(X%)\X%=X%-1
1240 IF T%=1 AND R%=0 THEN PRINT TAB(X%*3+3);"FAIL"
1250 IF T%=1 AND R%=1 THEN PRINT TAB(X%*3+3);"SUCCESS"
1260 IF X%<0 THEN RETURN
1270 GOTO 1200
1280 REM SCANNER
1290 C%=0\V%=0\VS=""
1300 IF LEN(SS)<>0 THEN 1310 ELSE 1340
1310 GOSUB 1350 \IF C$=" " GOTO 1300
1320 GOSUB 1390
1330 PRINT "C%="C%, "V%="V%,"VS="V$
1340 RETURN
1350 SS=CVT$$(SS,16%)
1360 C$=LEFT(SS,1)
1370 SS=RIGHT(SS,2)
1380 RETURN
1390 REM DETERMINE C$
1400 IF INSTR(1%,"0123456789",C$) THEN 1410 ELSE 1470
1410 VS=VS+CS
1420 V%=VAL(V$)
1430 IF LEN(SS) <>0 THEN 1440 ELSE 1460
1440 GOSUB 1350
```

```
1450 IF INSTR(1%,"0123456789",C$) THEN 1410
1460 C%=1\S$=C$+S$\RETURN
1470 REM KEY WORDS
1480 IF INSTR(1%,"ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789",C$) THEN 1490 ELSE
1490 V$=V$+C$\GOSUB 1350
1500 IF INSTR(1%,"ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789",C$) THEN 1490 ELSE
1510 IF LEN(S$)<> 0 THEN 1480 ELSE 1520
1520 FOR V%=1 TO 13
1530    IF K$(V%)=V$ THEN C%=3 ELSE 1550
        \S$=C$+S$
1540    RETURN
1550 NEXT V%
1560 V%=0%\C%=2\S$=C$+S$
1570 RETURN
1580 IF C$<>"<"THEN 1640
1590 V$=V$+C$
1600 GOSUB 1350
1610 IF C$="=" THEN V$=V$+C$\C%=4\V%=12\RETURN
1620 IF C$=">" THEN V$=V$+C$\C%=4\V%=13\RETURN
1630 C%=4\V%=2\S$=C$+S$\RETURN
1640 IF C$<>">" THEN 1690
1650 V$=V$+C$
1660 GOSUB 1350
1670 IF C$="=" THEN C%=4\V%=11\RETURN
1680 S$=C$+S$\C%=4\V%=1\RETURN
1690 IF C$<>"'" THEN 1750
1700 V$=V$+C$\Z%=1
1710 GOSUB 1350
1720 V$=V$+C$\Z%=Z%+1
1730 IF C$= "'" THEN C%=5\V%=0\Y%=Y%+1\Z%=Z%-2\RETURN
1740 GOSUB 1350\IF LEN(S$)<>0 GOTO 1720 ELSE PRINT "ERROR IN STRING"\RETUR
1750 IF INSTR(1%,"><=:;+-*/'!!!()",C$) THEN 1760 ELSE 1770
1760 V%=INSTR(1%,"><=:;+-*/'!!!()",C$) \C%=4\V$=V$+C$\RETURN
1770 V$=V$+C$\C%=0\V%=0
1780 RETURN
1790 REM SEMANTIC ROUTINES BEGIN HERE
1800 REM VARIABLE TABLE
1810 IF A%>1 THEN 1820 ELSE A$(A%)=V$\A%=A%+1\GOTO 1860
1820 B%=1
1830 IF A$(B%)=V$ THEN 1860
1840 B%=B%+1
1850 IF B%>A% THEN A$(A%)=V$\A%=A%+1\GOTO 1860
1855 GOTO 1830
1860 RETURN
1900 REM READ VARIABLE ROUTINE
1910 GOSUB 1800
1915 Y%=0
1916 IF I1%=1 THEN GOTO 2300
1917 Z$=W$+" CALL  TTYIN"\GOSUB 32000
     \Z$="         JMP  NUL2"\GOSUB 32000
1920 W$="      "
     \ Z$="TTYIN: IN  TTYST"\GOSUB 32000
     \Z$="       ANI  01"\GOSUB 32000
1940 Z$=W$+" JNZ  TTYIN"\GOSUB 32000
1950 Z$="      IN  TTYIO"\GOSUB 32000
     \Z$="       CMA"\GOSUB 32000
     \Z$="       ANI  07F"\GOSUB 32000
```

```
1970 Z$=W$+" STA  0102"\GOSUB 32000
1980 Z$="TTYSO: IN   TTYST"\GOSUB 32000
1990 Z$="       ANI  04"\GOSUB 32000
2000 Z$=W$+" JNZ  TTYSO"\GOSUB 32000
2010 Z$="       LDA  0102"\GOSUB 32000
     \Z$="       CMA"\GOSUB 32000
2020 Z$=W$+" OUT  TTYIO"\GOSUB 32000
2025 GOSUB 2400
2030 Z$=W$+" LXI  H,SAV"\GOSUB 32000
2040 Z$=W$+" MOV  A,M"\GOSUB 32000
2050 Z$="       CPI  1"\GOSUB 32000
2060 Z$=W$+" JZ   SEC"\GOSUB 32000
2070 Z$="       ADI  1"\GOSUB 32000
     \Z$="       MOV  M,A"\GOSUB 32000
     \Z$="ASCII: LXI  H,0102"\GOSUB 32000
2090 Z$= W$+" LXI  B,0100"\GOSUB 32000
2100 Z$=W$+" MOV  A,M"\GOSUB 32000
2120 Z$=W$+" RLC"\GOSUB 32000
2130 Z$=W$+" RLC"\GOSUB 32000
2140 Z$=W$+" RLC"\GOSUB 32000
2150 Z$=W$+" RLC"\GOSUB 32000
2160 Z$=W$+" STAX B"\GOSUB 32000
2170 Z$=W$+" JMP  TTYIN"\GOSUB 32000
2180 Z$=" SEC: LXI  H,0102"\GOSUB 32000
2190 Z$=W$+ " MOV  A,M"\GOSUB 32000
2200 Z$="       ANI  0F"\GOSUB 32000
2210 Z$=W$+" LXI  H,0100"\GOSUB 32000
     \Z$=W$+" ORA  M"\GOSUB 32000
     \Z$=W$+" STAX B"\GOSUB 32000
2234 Z$=W$+" MOV  A,E"\GOSU B 32000\Z$=W$+" CPI  1"\GOSUB 32000
2235 Z$=W$+" JZ   INXH"\GOSUB 32000
     \Z$="SECND: MVI  E,1"\GOSUB 32000
2240 Z$=W$+" LXI  H,SAV"\GOSUB 32000
2250 Z$=W$+" XRA  A"\GOSUB 32000
2260 Z$=W$+" MOV  M,A"\GOSUB 32000
2262 Z$=W$+" LDAX B"\GOSUB 32000
     \Z$=W$+" LXI  H,"+V$\GOSUB 32000
     \Z$=W$+" INX  H"\GOSUB 32000
     \Z$=W$+" MOV  M,A"\GOSUB 32000
2265 Z$=W$+" JMP  TTYIN"\GOSUB 32000
2270 Z$=" INXH: LXI  H,"+V$\GOSUB 32000
     \Z$=W$+" LDAX B"\GOSUB 32000
     \Z$=W$+" MOV  M,A"\GOSUB 32000
2275 Z$=W$+" LXI  H,SAV"\GOSUB 32000
     \Z$=W$+" XRA  A"\GOSUB 32000
     \Z$=W$+" MOV  M,A"\GOSUB 32000
2280 I1%=1
     \Z$="       RET"\GOSUB 32000
     \Z$="NUL2: MOV  A,A"\GOSUB 32000
     \RETURN
2300 Z$=W$+" CALL  TTYIN"\GOSUB 32000
     \W$="       "
     \RETURN
2400 Z$="       LDA  0102"\GOSUB 32000
     \Z$="       CPI  03A"\GOSUB 32000
     \Z$="       JC   NUMIN"\GOSUB 32000
     \Z$="       SUI  037"\GOSUB 32000
```

```
2410 Z$="      JMP  LTRIN"\GOSUB 32000
        \Z$="NUMIN: SUI  030"\GOSUB 32000
        \Z$="LTRIN: STA  0102"\GOSUB 32000
        \RETURN
2500 REM WRITE PARAMETER ROUTINE
2505 GOSUB 1800
2507 IF P1%=1 THEN GOTO 2900
2510 Z$=W$+" LXI  H,"+V$\GOSUB 32000
        \Z$="      INX  H"\GOSUB 32000
        \Z$="      CALL  TTYO"\GOSUB 32000
        \Z$="      JMP  NUL1"\GOSUB 32000
        \W$="      "
2520 Z$=" TTYO:  IN   TTYST"\GOSUB 32000
2530 Z$=W$+" ANI  04"\GOSUB 32000
2540 Z$=W$+" JNZ  TTYO"\GOSUB 32000
2550 Z$=W$+" MOV  A,M"\GOSUB 32000
2560 Z$=W$+" ANI  240"\GOSUB 32000
2570 Z$=W$+" RRC"\GOSUB 32000
2580 Z$=W$+" RRC"\GOSUB 32000
2590 Z$=W$+" RRC"\GOSUB 32000
2600 Z$=W$+" RRC"\GOSUB 32000
2610 Z$=W$+" CPI  0A"\GOSUB 32000
2620 Z$=W$+" JP   LTR1"\GOSUB 32000
2630 Z$=W$+" ADI  030"\GOSUB 32000
2640 Z$=W$+" JMP  TTO"\GOSUB 32000
2650 Z$="LTR1: ADI  037"\GOSUB 32000
2660 Z$="TTO: CMA"\GOSUB 32000
2670 Z$=W$+" OUT  TTYIO"\GOSUB 32000
2680 Z$=" TT02: IN  TTYST"\GOSUB 32000
2690 Z$=W$+" ANI  04"\GOSUB 32000
2700 Z$=W$+" JNZ  TT02"\GOSUB 32000
2710 Z$=W$+" MOV  A,M"\GOSUB 32000
2720 Z$=W$+" ANI  0F"\GOSUB 32000
2730 Z$=W$+" CPI  0A"\GOSUB 32000
2740 Z$="        JP   LTR2"\GOSUB 32000
2750 Z$=W$+" ADI  030"\GOSUB 32000
2760 Z$=W$+" JMP  OT"\GOSUB 32000
2770 Z$="  CHK: LXI  H,SAV"\GOSUB 32000
        \Z$="      MOV  A,M"\GOSUB 32000
2790 Z$=W$+" CPI  1"\GOSUB 32000
2800 Z$=W$+"  JZ  OTRT"\GOSUB 32000
2810 Z$=W$+" ADI  1"\GOSUB 32000
2820 Z$=W$+" MOV  M,A"\GOSUB 32000
2830 Z$=W$+" LXI  H,"+V$\GOSUB 32000
2840 Z$=W$+" JMP  TTYO"\GOSUB 32000
2860 Z$= "LTR2: ADI  037"\GOSUB 32000
2870 Z$=" OT: CMA"\GOSUB 32000
2880 Z$=W$+" OUT  TTYIO"\GOSUB 32000
2885 P1%=1
        \Z$="      JMP  CHK"\GOSUB 32000
        \Z$=" OTRT: RET"\GOSUB 32000
        \Z$=" NUL1: MOV  A,A"\GOSUB 32000
2890 RETURN
2900 Z$=W$+" LXI  H,"+V$\GOSUB 32000
        \Z$="       CALL TTYO"\GOSUB 32000
        \W$="      "
        \RETURN
```

```
3000 REM STRING ROUTINE
3010 IF D%>1 THEN 3020
3012 D1$=NUM1$(D%)
3014 D$(D%)="STG"+D1$\D%=D%+1
3016 D$(D%)=V$\D%=D%+1\GOTO 3070
3020 B%=2
3030 IF D$(B%)=V$ THEN 3080
3040 B%=B%+1
3050 IF B%>D% THEN 3052 ELSE 3060
3052 D1$=NUM1$(D%)\D$(D%)="STG"+D1$\D%=D%+1
3054 D$(D%)=V$\D%=D%+1\B%=D%-1\GOTO 3080
3060 GOTO 3030
3070 B%=2
3080 B%=B%-1
3090 Z$=W$+" LXI  H,"+D$(B%)\GOSUB 32000
3095 W$="      "
3100 Z$=W$+" MVI  B,"+NUM1$(Z%)\GOSUB 32000
3101 IF D%=3 THEN GOTO 3102 ELSE Z$=W$+" CALL  PRINT"\GOSUB 32000\GOTO 3250
3102 Z$=W$+" CALL PRINT"\GOSUB 32000
     \Z$="       JMP  NUL6"\GOSUB 32000
3110 Z$=" PRINT: MOV  C,M"\GOSUB 32000
3120 Z$=W$+" CALL  TTYOUT"\GOSUB 32000
3130 Z$=W$+" INX  H"\GOSUB 32000
3140 Z$=W$+" DCR  B"\GOSUB 32000
3150 Z$=W$+" JNZ  PRINT"\GOSUB 32000
3160 Z$=W$+" JMP  ENDPT"\GOSUB 32000
3170 Z$="TTYOUT: IN   TTYST"\GOSUB 32000
3180 Z$=W$+" ANI  04"\GOSUB 32000
3190 Z$=W$+" JNZ  TTYOUT"\GOSUB 32000
3200 Z$=W$+" MOV  A,C"\GOSUB 32000
3210 Z$=W$+" CMA"\GOSUB 32000
3220 Z$=W$+" OUT  00"\GOSUB 32000
3230 Z$=W$+" RET"\GOSUB 32000
3240 Z$="ENDPT: RET"\GOSUB 32000
     \Z$=" NUL6: MOV  A,A"\GOSUB 32000
3250 RETURN
3500 REM GOTO ROUTINE
3520 Z$=W$+" JMP  "+V$\GOSUB 32000
3530 W$="      "
3540 RETURN
3700 REM GOSUB ROUTINE
3720 Z$=W$+" CALL "+V$\GOSUB 32000
3730 W$="      "
3740 RETURN
3800 REM RETURN ROUTINE
3810 Z$=W$+" RET "\GOSUB 32000
3820 W$="      "
3830 RETURN
4000 REM "MOVE" TO ROUTINE
4010 IF C%=1 THEN GOTO 4100
4020 GOSUB 1800
4030 Z$=W$+" LXI  H,"+V$\GOSUB 32000
4040 W$="      "
4050 Z$=W$+" MOV  E,M"\GOSUB 32000
4060 Z$=W$+" INX  H"\GOSUB 32000
4070 Z$=W$+" MOV  D,M"\GOSUB 32000
4080 GOTO 4120
```

```
4100 Z$=W$+" LXI  D,"+V$\GOSUB 32000
4110 W$="      "
4120 RETURN
4200 REM MOVE "TO" ROUTINE
4210 GOSUB 1800
4220 Z$=WS+" LXI  H,"+V$\GOSUB 32000
4230 Z$=WS+" MOV  A,E"\GOSUB 32000
4240 Z$=WS+" MOV  M,A"\GOSUB 32000
4250 Z$=WS+" INX  H"\GOSUB 32000
4260 Z$=WS+" MOV  A,D"\GOSUB 32000
4270 Z$=WS+" MOV  M,A"\GOSUB 32000
4280 RETURN
4500 REM LABEL ROUTINE
4510 W$=V$+":"
4520 RETURN
4700 REM IF THEN ELSE ROUTINE
4710 Z$=W$+" LXI H,RESL1"\GOSUB 32000
4720 W$="      "
4730 Z$=W$+" MOV  A,M"\GOSUB 32000
4740 Z$=W$+" MOV  E,A"\GOSUB 32000
4750 Z$=W$+" INX  H"\GOSUB 32000
4760 Z$=W$+" MOV  A,M"\GOSUB 32000
4770 Z$=W$+" MOV  D,A"\GOSUB 32000
4775 Z$=W$+" PUSH  D"\GOSUB 32000
4780 RETURN
4810 W%=V%
4820 RETURN
4825 REM CONDITION ROUTINE
4827 Z$=W$+" POP  D"\GOSUB 32000\W$="      "
4830 IF W%=1 THEN GOSUB 5200
4835 IF W%=2 THEN GOSUB 5300
4840 IF W%=3 THEN GOSUB 5500
4845 IF W%=11 THEN GOSUB 5600
4850 IF W%=12 THEN GOSUB 5800
4855 IF W%=13 THEN GOSUB 5100
4860 RETURN
4900 REM CONDITION ROUTINE
5100 Z$=W$+" LXI  H,RESL1"\GOSUB 32000
     \Z$="      INX  H"\GOSUB 32000
     \Z$="      MOV  A,M"\GOSUB 32000
     \Z$="      CMP  D"\GOSUB 32000
5110 Z$="    JNZ  THN"+NUM1$(T1%)\GOSUB 32000
     \Z$="      DCX  H"\GOSUB 32000
     \Z$="      MOV  A,M"\GOSUB 32000
     \Z$="      CMP  E"\GOSUB 32000
     \Z$="      JNZ  THN"+NUM1$(T1%)\GOSUB 32000
     \Z$="      JMP  ELS"+NUM1$(E1%)\GOSUB 32000
5160 RETURN
5200 Z$=WS+" LXI  H,RESL1"\GOSUB 32000
     \Z$=WS+" INX  H"\GOSUB 32000
     \Z$=WS+" MOV  A,M"\GOSUB 32000
     \Z$=WS+" CMP  D"\GOSUB 32000
     \Z$=WS+" JC  THN"+NUM1$(T1%)\GOSUB 32000
     \Z$=WS+" DCX  H"\GOSUB 32000
     \Z$=WS+" MOV  A,M"\GOSUB 32000
     \Z$=WS+" CMP  E"\GOSUB 32000
     \Z$=WS+" JC  THN"+NUM1$(T1%)\GOSUB 32000
```

```
        \Z$=W$+" JMP  ELS"+NUM1$(E1%)\GOSUB 32000
5290 RETURN
5300 Z$=W$+" LXI  H,RESL1"\GOSUB 32000
        \Z$="        INX  H"\GOSUB 32000
        \Z$="        MOV  A,M"\GOSUB 32000
        \Z$="        CMP  D"\GOSUB 32000
        \Z$="        JC   ELS"+NUM1$(E1%)\GOSUB 32000
5310 Z$="        JNZ  THN"+NUM1$(T1%)\GOSUB 32000
        \Z$="        DCX  H"\GOSUB 32000
        \Z$="        MOV  A,M"\GOSUB 32000
        \Z$="        CMP  E"\GOSUB 32000
                \Z$="        JC   ELS"+NUM1$(E1%)\GOSUB 32000
        \Z$="        JZ   ELS"+NUM1$(E1%)\GOSUB 32000
        \Z$="        JMP  THN"+NUM1$(T1%)\GOSUB 32000
5420 RETURN
5500 Z$=W$+" LXI  H,RESL1"\GOSUB 32000
        \Z$="        INX  H"\GOSUB 32000
        \Z$="        MOV  A,M"\GOSUB 32000
        \Z$="        CMP  D"\GOSUB 32000
        \Z$="        JNZ  ELS"+NUM1$(E1%)\GOSUB 32000
        \Z$="        DCX  H"\GOSUB 32000
5510 Z$="        MOV  A,M"\GOSUB 32000
        \Z$="        CMP  E"\GOSUB 32000
        \Z$="        JNZ  ELS"+NUM1$(E1%)\GOSUB 32000
        \Z$="        JMP  THN"+NUM1$(T1%)\GOSUB 32000
5595 RETURN
5600 Z$=W$+" LXI  H,RESL1"\GOSUB 32000
        \Z$="        INX  H"\GOSUB 32000
        \Z$="        MOV  A,M"\GOSUB 32000
        \Z$="        CMP  D"\GOSUB 32000
5605 Z$="        JC   THN"+NUM1$(T1%)\GOSUB 32000
        \Z$="        JNZ  ELS"+NUM1$(E1%)\GOSUB 32000
        \Z$="        DCX  H"\GOSUB 32000
5610 Z$="        MOV  A,M"\GOSUB 32000
        \Z$="        CMP  E"\GOSUB 32000
        \Z$="        JZ   THN"+NUM1$(T1%)\GOSUB 32000
        \Z$="        JC   THN"+NUM1$(T1%)\GOSUB 32000
        \Z$="        JMP  ELS"+NUM1$(E1%)\GOSUB 32000
5710 RETURN
5800 Z$=W$+" LXI  H,RESL1"\GOSUB 32000
        \Z$="        INX  H"\GOSUB 32000
        \Z$="        MOV  A,M"\GOSUB 32000
        \Z$="        CMP  D"\GOSUB 32000
        \Z$="        JC   ELS"+NUM1$(E1%)\GOSUB 32000
5810 Z$="        DCX  H"\GOSUB 32000
        \Z$="        MOV  A,M"\GOSUB 32000
        \Z$="        CMP  E"\GOSUB 32000
        \Z$="        JZ   THN"+NUM1$(T1%)\GOSUB 32000
        \Z$="        JC   ELS"+NUM1$(E1%)\GOSUB 32000
        \Z$="        JMP  THN"+NUM1$(T1%)\GOSUB 32000
5895 RETURN
6000 REM THEN ROUTINE
6010 W$="THN"+NUM1$(T1%)+":"
        \T1%=T1%+1
        \RETURN
6100 REM ELSE ROUTINE
6110 W$="ELS"+NUM1$(E1%)+":"
```

```
             \E1%=E1%+1
             \Z$="      JMP  NXT"+NUM1$(E1%)
             \GOSUB 32000
             \RETURN
6200 Z$=" NXT"+NUM1$(E1%)+": MOV  A,A"\GOSUB 32000
6210 RETURN
6500 REM END ROUTINE
6510 Z$=W$+" JMP  03800"\GOSUB 32000\W$="     "\RETURN
7000 REM EXPRESSION ROUTINE
7010 IF C%=1 THEN GOTO 7020 ELSE GOSUB 1800
7020 E2$(E2%)=V$
             \E2%=E2%+1
7030 IF S1%=1 THEN GOTO 7500
7040 RETURN
7100 E2$(E2%)=V1$
             \E2%=E2%+1
7120 RETURN
7200 E2$(E2%)=V1$
7210 E2%=E2%+1
7220 S1%=1
7225 RETURN
7500 E5%=E2%
             \IF V1$="*" THEN GOSUB 10000
7505 IF V1$="/" THEN GOSUB 12000
7510 E2$(E2%)=" "
             \E2$(E2%-1)=" "
             \E2$(E2%-2)=" "
             \E2%=E2%-3
7540 E2$(E2%)="RESL"+NUM1$(R1%)
7550 R1%=R1%+1
             \E2%=E2%+1
             \S1%=0
7560 RETURN
7600 E2$(E2%)=V1$
7605 E3%=E2%
7610 E2%=E2%+1
7620 RETURN
7700 E5%=E3%+2
7710 E5%=E5%+2
7715 IF E5%>E2% THEN GOTO 7800
7720 IF E2$(E5%-2)="+" THEN GOSUB 9000\GOTO 7740
7730 IF E2$(E5%-2) ="-" THEN GOSUB 9500 ELSE GOTO 7800
7740 E2$(E5%-1) ="RESL"+NUM1$(R1%)
             \GOTO 7710
7800 E2%=E3%
             \E2$(E2%)="*"
             \E2%=E2%+1
             \E2$(E2%)="RESL"+NUM1$(R1%)
             \E2%=E2%+1
             \E5%=E2%
             \GOSUB 10000
             \GOSUB 7510
7835 RETURN
7900 IF E2%<>2 THEN GOTO 7905
             \Y%=A%-1
             \IF A$(Y%)=E2$(E2%-1) THEN GOTO 7902
             \Y%=Y%-1
```

```
        \IF Y%=0 THEN Z$=W$+" LXI  D,"+E2$(E2%-1)\GOSUB 32000
        \W$="      "
        \GOTO 7950
7902 Z$=W$+" LXI  H,"+E2$(E2%-1)\GOSUB 32000
        \W$="      "
        \Z$=W$+" MOV  E,M"\GOSUB 32000
        \Z$=W$+" INX  H"\GOSUB 32000
        \Z$=W$+" MOV  D,M"\GOSUB 32000
        \GOTO 7950
7905 E5%=2
7907 E5%=E5%+2
        \IF E5%>E2% THEN GOTO 7950
7910 IF E2$(E5%-2)="+" THEN GOSUB 9000\GOTO 7930
7920 IF E2$(E5%-2)="-" THEN GOSUB 9500 ELSE GOTO 7950
7930 E2$(E5%-1)="RESL"+NUM1$(R1%)
7940 GOTO 7907
7950 Z$=W$+" LXI  H,RESL1"\GOSUB 32000
        \W$="      "
        \Z$=W$+" MOV  M,E"\GOSUB 32000
        \Z$=W$+" INX  H"\GOSUB 32000
7955 Z$=W$+" MOV M,D"\GOSUB 32000
7960 E2%=1\R1%=1\S1%=0
        \RETURN
8000 RETURN
9000 REM ADD INSTRUCTIONS TO LOAD REGISTERS
9010 GOSUB 25000
9200 Z$=W$+" XCHG"\GOSUB 32000
        \Z$="       DAD  B"\GOSUB 32000
        \W$="      "
9210 Z$=W$+" XCHG"\GOSUB 32000
        \Z$="       LXI  H,RESL"+NUM1$(R1%)\GOSUB 32000
        \Z$="       MOV  M,E"\GOSUB 32000
        \Z$="       INX  H"\GOSUB 32000
9220 Z$=W$+" MOV  M,D"\GOSUB 32000
        \E2$(E2%+1)="RESL"+NUM1$(R1%)
9229 RETURN
9500 REM SUBTRACT INSTRUCTION/MUST LOAD REGISTERS
9510 GOSUB 25000
9600 IF S2%=1 THEN GOTO 9910
9610 Z$=W$+" CALL  SUBT"\GOSUB 32000
        \Z$="       JMP  NUL3"\GOSUB 32000

9700 Z$="SUBT: MOV  A,E"\GOSUB 32000
9705 W$="      "
9710 Z$=W$+" CMA"\GOSUB 32000
9720 Z$=W$+" ADI  1"\GOSUB 32000
9725 Z$=W$+" MOV  E,A"\GOSUB 32000
9730 Z$=W$+" JC   CARRY"\GOSUB 32000
9750 Z$=W$+" JMP  SECND"\GOSUB 32000
9760 Z$="CARRY: MOV  A,D"\GOSUB 32000
9770 Z$=W$+" CMA"\GOSUB 32000
9780 Z$=W$+" ADI  1"\GOSUB 32000
9790 Z$=W$+" MOV  D,A"\GOSUB 32000
9800 Z$=W$+" JMP  DADD"\GOSUB 32000
9810 Z$="SECND: MOV  A,D"\GOSUB 32000
9820 Z$=W$+" CMA"\GOSUB 32000
9830 Z$=W$+" MOV  D,A"\GOSUB 32000
```

```
9840 Z$=" DADD: XCHG"\GOSUB 32000
     \Z$="      DAD  B"\GOSUB 32000
     \GOSUB 9210
9860 S2%=1
     \Z$="       RET"\GOSUB 32000
     \Z$="NUL3: MOV  A,A"\GOSUB 32000
9900 RETURN
9910 Z$=W$+" CALL SUBT"\GOSUB 32000
     \RETURN
10000 REM MULTIPLICATION ROUTINE
10002 GOSUB 25000
10006 IF M1%=1 THEN GOTO 10700
10007 Z$=W$+" CALL MULT"\GOSUB 32000
     \Z$="      JMP  NUL4"\GOSUB 32000
10200 W$="      "
     \Z$=" MULT: MVI  B,0"\GOSUB 32000
     \Z$="       MOV  D,E"\GOSUB 32000
10210 Z$=W$+" MVI  E,9"\GOSUB 32000
10220 Z$="MULT0: MOV  A,C"\GOSUB 32000
10230 Z$=W$+" RAR"\GOSUB 32000
10240 Z$=W$+" MOV  C,A"\GOSUB 32000
10250 Z$=W$+" DCR  E"\GOSUB 32000
10260 Z$=W$+" JZ   DONE"\GOSUB 32000
10270 Z$=W$+" MOV  A,B"\GOSUB 32000
10280 Z$=W$+" JNC  MULT1"\GOSUB 32000
10290 Z$=W$+" ADD  D"\GOSUB 32000
10300 Z$="MULT1: RAR"\GOSUB 32000
10310 Z$=W$+" MOV  B,A"\GOSUB 32000
10320 Z$="      JMP  MULT0"\GOSUB 32000
     \M1%=1
     \Z$=" DONE: RET"\GOSUB 32000
     \Z$=" NUL4: MOV  A,A"\GOSUB 32000
10330 Z$="      LXI  H,RESL"+NUM1$(R1%)\GOSUB 32000
     \Z$="      MOV  M,C"\GOSUB 32000
     \Z$="      INX  H"\GOSUB 32000
     \Z$="      MOV  M,B"\GOSUB 32000
     \E2$(E2%-1)="RESL"+NUM1$(R1%)
10500 RETURN
10700 Z$=W$+" CALL MULT"\GOSUB 32000
     \GOSUB 10330
     \RETURN
12000 REM DIVISION ROUTINE
12002 GOSUB 25000
12006 IF D2%=1 THEN GOTO 12700
12007 Z$=W$+" CALL  DIV"\GOSUB 32000
     \Z$="      JMP  NUL5"\GOSUB 32000
12010 W$="      "
12200 Z$="  DIV: MOV  A,D"\GOSUB 32000
12210 Z$=W$+" CMA"\GOSUB 32000
12220 Z$=W$+" MOV  D,A"\GOSUB 32000
12230 Z$=W$+" MOV  A,E"\GOSUB 32000
12240 Z$=W$+" CMA"\GOSUB 32000
12250 Z$=W$+" MOV  E,A"\GOSUB 32000
12260 Z$=W$+" INX  D"\GOSUB 32000
12270 Z$="      MOV  L,C"\GOSUB 32000
     \Z$="      MOV  H,B"\GOSUB 32000
     \Z$="      LXI  B,0"\GOSUB 32000
```

```
12280 Z$="DIVDE: DAD  D"\GOSUB 32000
      \Z$="       MOV  A,H"\GOSUB 32000
      \Z$="       RAL"\GOSUB 32000
      \Z$="       JC  FNDIV"\GOSUB 32000
12290 Z$="      INX  B"\GOSUB 32000
      \Z$="       JMP  DIVDE"\GOSUB 32000
      \Z$="FNDIV: XRA  A"\GOSUB 32000
12570 D2%=1
      \Z$="        RET"\GOSUB 32000
      \Z$=" NUL5: MOV  A,A"\GOSUB 32000
12600 Z$=W$+" LXI  H,RESL"+NUM1$(R1%)\GOSUB 32000
      \Z$="        MOV  M,C"\GOSUB 32000
      \Z$="        INX  H"\GOSUB 32000
      \Z$="        MOV  M,B"\GOSUB 32000
12620 E2$(E2%-1)="RESL"+NUM1$(R1%)
12650 RETURN
12700 Z$=W$+" CALL  DIV"\GOSUB 32000
      \GOSUB 12600
      \RETURN
15000 REM COMPUTE ROUTINE
15010 GOSUB 1800
      \Z$=W$+" LXI  H,RESL1"\GOSUB 32000
      \W$="      "
15015 Z$=W$+" MOV  D,M"\GOSUB 32000
      \Z$=W$+" INX  H"\GOSUB 32000
      \Z$=W$+" MOV  E,M"\GOSUB 32000
15020 Z$=W$+" LXI  H,"+V$\GOSUB 32000
      \Z$=W$+" MOV  M,D"\GOSUB 32000
      \Z$=W$+" INX  H"\GOSUB 32000
      \Z$=W$+" MOV  M,E"\GOSUB 32000
      \RETURN
15100 Z$=W$+" LXI  H,RESL1"\GOSUB 32000
      \Z$=W$+" MOV  D,M"\GOSUB 32000
      \Z$=W$+" INX  H"\GOSUB 32000
      \Z$=W$+" MOV  E,M"\GOSUB 32000
20000 REM ROUTINE TO BUILD ASSEMBLY LANGUAGE FILE
20010 W$="      "
20110 Z$="       ORG  0100"\GOSUB 32100
      \Z$="        BYTE  00"\GOSUB 32100
      \Z$="        BYTE  00"\GOSUB 32100
      \Z$="        BYTE  00"\GOSUB 32100
20115 R1%=9
20120 IF R1%=0 THEN B%=1\GOTO 20140
20130 Z$="RESL"+NUM1$(R1%)+": WORD  00"\GOSUB 32100
      \R1%=R1%-1\GOTO 20120
20140 IF (D%=1 OR D%=0) THEN GOTO 20170
20150 Z$=D$(B%)+": TXT  "+D$(B%+1)\GOSUB 32100
20160 B%=B%+2\D%=D%-2\GOTO 20140
20170 A%=A%-1
20180 IF A%<1 THEN GOTO 20205
20190 Z$=A$(A%)+": WORD  00"\GOSUB 32100 \GOTO 20170
20205 Z$=" SAV: BYTE 00"\GOSUB 32100
      \Z$="TTYST: EQU  01"\GOSUB 32100
      \Z$="TTYIO: EQU  00"\GOSUB 32100
20250 ON ERROR GOTO 26000
20300 INPUT LINE #2%,Z$\GOSUB 32100
20310 IF Z$="      END" THEN GOTO 20400 ELSE GOTO 20300
```

```
20400 RETURN
21130 Z$="RESL"+NUM1$(R1%)+" WORD  00"\GOSUB 32100\
        R%=R%-1\GOTO 20120
25000 E4%=E5%-1
25002 FOR Y%= 1 TO 20
25004 IF A$(Y%)= E2$(E4%) THEN GOTO 25100
25006 NEXT Y%
25010 FOR Y%= 1 TO 9
        \IF E2$(E4%)="RESL"+NUM1$(Y%) THEN GOTO 25100
25020 NEXT Y%
25030 Z$=W$+" LXI  D,"+E2$(E4%)\GOSUB 32000
25035 W$="     "
25040 E4%=E5%-3
25042 FOR Y%= 1 TO 20
25044 IF A$(Y%)= E2$(E4%) THEN GOTO 25200
25046 NEXT Y%
25050 FOR Y%= 1 TO 9
25052 IF E2$(E4%)="RESL"+NUM1$(Y%) THEN GOTO 25200
25054 NEXT Y%
25060 Z$=W$+" LXI  B,"+E2$(E4%)\GOSUB 32000\GOTO 25300
25100 Z$=W$+" LXI  H,"+E2$(E4%)\GOSUB 32000
25110 W$="     "
        \Z$=W$+" MOV  E,M"\GOSUB 32000
        \Z$=W$+" INX  H"\GOSUB 32000
        \Z$=W$+" MOV  D,M"\GOSUB 32000
        \GOTO 25040
25200 Z$=W$+" LXI  H,"+E2$(E4%)\GOSUB 32000
        \W$="     "
        \Z$=W$+" MOV  C,M"\GOSUB 32000
        \Z$=W$+" INX  H"\GOSUB 32000
        \Z$=W$+" MOV  B,M"\GOSUB 32000
25300 RETURN
26000  RESUME 20400
32000 REM PRINT FILE ROUTINE
32010 PRINT #2% USING Z$
32020 RETURN
32100 PRINT #1% USING Z$
32150 RETURN
32500 REM SEMANTIC DECODE HERE
32510 IF P%=64 THEN GOSUB 7900\GOSUB 4825\GOSUB 6000\RETURN
32520 IF P%=72 THEN GOSUB 6100\RETURN
32530 IF P%=92 THEN GOSUB 4700\RETURN
32540 IF P%=132 THEN GOSUB 1900\RETURN
32550 IF P%=140 AND C%=2 THEN GOSUB 2500\RETURN
32560 IF P%=144 THEN GOSUB 3000 \RETURN
32570 IF P%=152 THEN GOSUB 3500\RETURN
32580 IF P%=160 THEN GOSUB 3700 \RETURN
32590 IF P%=168 THEN GOSUB 4000\RETURN
32600 IF P%=176 THEN GOSUB 4200\RETURN
32610 IF P%=180 AND V%=8 THEN GOSUB 3800\RETURN
32620 IF P%=196 THEN GOSUB 4500\RETURN
32630 IF (P%=256 OR P%=348) THEN GOSUB 7600\RETURN
32640 IF (P%=260 OR P%=352) THEN GOSUB 7700\RETURN
32650 IF P%=264 AND C%=1 THEN GOSUB 7000\RETURN
32652 IF P%=268 AND C%=2 THEN GOSUB 7000\RETURN
32654 IF (P%=272 OR P%=284) THEN V1$=V$\RETURN
32660 IF (P%=224) THEN GOSUB 7100\RETURN
```

```
32670 IF P%=240 THEN GOSUB 7200\RETURN
32680 IF P%=328 THEN GOSUB 15000\RETURN
32690 IF P%=324 THEN GOSUB 7900\RETURN
32692 IF P%=356 AND C%=3 THEN GOSUB 6500\RETURN
32695 IF P%=88 THEN GOSUB 7900\GOSUB 4810
32700 RETURN
32760 END
```

END